

Université de Mons UMONS



MANUEL d'utilisation de *Processing*

Mallette CYBERPACK

Edition 2013

Résumé :

Le projet Cyberpack vise à développer la créativité à travers la mise en œuvre de projets basés sur les technologies numériques utilisées quotidiennement par les élèves (son, image, 3D, web, facebook, etc.). Il a également pour objectif d'encadrer les professeurs qui mettront en œuvre ces projets, par exemple dans le cadre des cyber-classes.

Le projet consiste en la création d'outils pédagogiques pour l'apprentissage progressif de Processing afin qu'enseignants et élèves deviennent créateurs d'applications et de contenus numériques. Parmi ces outils :

1. une mallette pédagogique à l'attention des élèves du secondaire pour développer leur créativité. La mallette comprendra une introduction au logiciel Processing (gratuit et utilisé par de jeunes artistes numériques dans le monde entier) et un guide pour la réalisation de projets numériques, et le matériel pour les réaliser ;
 2. une plateforme d'apprentissage en ligne, utilisant les réseaux sociaux pour le partage d'expériences entre écoles (questions, réalisations, vidéos de démonstrations, etc.).
-

Partenaires

Le projet *Cyberpack* est porté par diverses composantes de l'UMONS :

- L'Institut UMONS/Numediart, Institut pour les Technologies des Arts Numériques porte le projet et apporte ses compétences techniques en matière d'innovations technologiques. En effet, il présente l'originalité de développer des projets et des produits innovants pour le monde artistique. L'équipe de Numédiart fait preuve d'une expertise de longues dates dans les technologies numériques liées au traitement des signaux et des images et compte plusieurs brevets à son actif notamment dans le domaine des technologies vocales. L'équipe a également une forte compétence pédagogique puisqu'elle enseigne aux futurs ingénieurs et met en œuvre à cet effet des techniques pédagogiques innovantes.
- ApplicaSciences et le Carré des sciences sont les cellules de diffusion des sciences et techniques de l'UMONS. Elles ont à cœur de sensibiliser le public aux innovations technologiques et aux recherches de pointe et organisent des activités visant à développer la créativité des jeunes du secondaire (stages de réalisation de projets concrets □). Leurs activités récurrentes sont reprises sur leurs sites web respectifs www.umons.ac.be/applicasciences et <http://cds.umons.ac.be>. Elles disposent à elles-deux d'un solide réseau de contacts de professeurs du secondaire. Le Carré des sciences a une expertise certaine dans le développement de maquettes pédagogiques pour la Communauté Française en collaboration avec le service du professeur Marc Demeuse, chef du service Méthodologie et Formations.
- Le professeur Bruno Le Lièvre apportera son expertise au niveau de la création du site web permettant aux utilisateurs (professeurs et élèves du secondaire) d'accéder à la créativité et de développer des produits innovants étape par étape.
- ARTS2, résultant de la fusion récente de l'ESAPV (arts plastiques et visuels, Mons) et du conservatoire de Mons fournit au projet des compétences artistiques dans le domaine du monde numérique. ARTS2 fait également partie du consortium NUMEDIART, qui regroupe 15 acteurs industriels, éducatifs et culturels liés au développement des technologies des arts numériques. ARTS2 a accepté de contribuer au projet.

Remerciements

Le projet *Cyberpack* a été sélectionné dans le cadre de l'appel à projets *Creative People*¹ lancé par le programme *Creative Wallonia*². Nous remercions *Creative Wallonia* et l'Agence de Stimulation Economique (ASE)³ de soutenir notre projet.

Ce manuel d'utilisation de *Processing* n'est que la remise en page du manuel *flossmanuals*⁴. Nous tenons à remercier la communauté pour la mise à disposition de ce document.

1. <http://www.creative-wallonia.be/creative-people/>

2. <http://www.creative-wallonia.be>

3. <http://www.as-e.be/>

4. <http://fr.flossmanuals.net/>

Table des matières

1	Qu'est-ce que <i>Processing</i> ?	1
1.1	Présentation	1
1.2	Les ressources sur <i>processing.org</i>	2
1.3	Autres ressources disponibles	2
1.4	Extensions et projets alternatifs	3
2	Installation de <i>Processing</i>	5
2.1	Sous Windows	5
2.2	Sous Mac OS X	5
2.3	Sous GNU/Linux	6
3	Introduction au logiciel <i>Processing</i>	9
3.1	Les bases du langage	9
3.1.1	L'interface	9
3.1.2	Premières clés	10
3.2	L'espace de dessin	12
4	Dessiner avec <i>Processing</i>	13
4.1	Les formes	13
4.2	Les couleurs	19
4.3	Le texte	23
4.4	Les images	25
4.5	Les styles de bordures	32
4.6	La typographie	34
4.7	Les transformations	38

5 Programmer avec <i>Processing</i>	47
5.1 Les variables	47
5.2 Les conditions	51
5.3 Les répétitions	53
5.4 Les listes	57
5.5 Les méthodes	63
5.6 Les objets	69
5.7 Les commentaires	73
6 Animer avec <i>Processing</i>	75
6.1 La méthode draw	75
6.2 La ligne de temps	80
6.3 L'animation d'un objet	83
6.4 L'animation de plusieurs objets	88
6.5 La lecture du son	93
7 Interagir avec <i>Processing</i>	99
7.1 Les évènements clavier	99
7.2 Les évènements souris	102
7.3 L'entrée microphone	106
7.4 L'entrée vidéo	110
8 Exposer ses projets <i>Processing</i>	113
8.1 L'exportation	113
8.2 L'impression	115
8.3 La vidéo	118
A Erreurs & Astuces	121
A.1 Les erreurs courantes	121
A.2 Les astuces	124

Table des matières	ix
B Pour étendre les fonctionnalités de <i>Processing</i>	129
B.1 Les bibliothèques externes	129
Index	132

Qu'est-ce que *Processing* ?

Sommaire

1.1 Présentation	1
1.2 Les ressources sur <i>processing.org</i>	2
1.3 Autres ressources disponibles	2
1.4 Extensions et projets alternatifs	3

1.1 Présentation

Génèse

Processing a été créé en 2001 par Ben Fry¹ et Casey Reas². Tous deux étudiants aux Media Lab du Massachusetts Institute of Technology³, sous la tutelle de John Maeda⁴ et en poursuite de son projet *Design by Numbers* qui est le titre d'un livre, ainsi que d'un programme éponyme⁵.

1. http://benfry.com/http://twitter.com/ben_fry
2. <http://reas.com/http://twitter.com/reas/>
3. http://www.mit.edu/http://fr.wikipedia.org/wiki/Massachusetts_Institute_of_Technology
4. http://fr.wikipedia.org/wiki/John_Maedahttps://twitter.com/johnmaeda
5. <http://www.amazon.com/Design-Numbers-John-Maeda/dp/0262632446>
6. <http://processing.org/about/>

Description

Processing est un langage de programmation et un environnement open source pour les personnes désirant créer des images, des animations et des interactions. Originellement conçu comme software de prototypage et pour enseigner les bases de la programmation dans un contexte visuel, *Processing* s'est également étendu en un outil pour générer des travaux finis. Aujourd'hui, de nombreux étudiants, artistes, designers, chercheurs, et passionnés utilisent

Processing pour apprendre, prototyper, et produire⁶.

1.2 Les ressources sur *processing.org*

- Présentation de l'environnement de développement de Processing :
<http://processing.org/reference/environment/>
- Tutoriaux pour apprendre à utiliser Processing (en anglais) :
<http://processing.org/learning/>
- L'explication du langage :
<http://processing.org/reference/> ;
par ordre alphabétique <http://processing.org/reference/alpha.html>.
Également directement accessible directement dans la fenêtre en sélectionnat et click droit *Find in reference*.
Un wiki est également disponible : http://wiki.processing.org/w/Main_Page
- Des exemples d'utilisation :
<http://processing.org/learning/basics/>
<http://processing.org/learning/topics/>
<http://processing.org/learning/3d/>
- Les Librairies, directement intégrées dans Processing et les contributions à installer :
<http://processing.org/reference/libraries/>

- Des livres en anglais :
<http://processing.org/learning/books/>
- Pour étendre les capacités de Processing :
<http://processing.org/reference/tools/>
- Un forum d'entre-aide, sources de support technique
<http://forum.processing.org/>
<http://answers.oreilly.com/tag/processing/>
- Des projets mis en oeuvre avec *Processing* :
<http://processing.org/exhibition/>
OpenProcessing : <http://www.openprocessing.org/>

1.3 Autres ressources disponibles

Floss Manuals

Floss Manuals⁷ est une plateforme de partage et de co-crédation plurilingue. Floss Manuals met à la disposition du grand public et des professionnels une collection de livres en français sur les logiciels, la culture libre et le matériel libre. S'appuyant sur la communauté d'utilisateurs de ces outils, notamment les auteurs, éditeurs, artistes, développeurs de logiciels et activistes, ces livres sont consultables librement en différents formats (pages web, pdf et epub).

7. <http://fr.flossmanuals.net/>

Ressources et support technique

En ligne Beaux Arts d'Aix-en-Provence, France :

<http://www.ecole-art-aix.fr/rubrique81.html>,

<http://www.ecole-art-aix.fr/rubrique50.html>

Codelab :

<http://codelab.fr/39>

Learning Processing :

<http://www.learningprocessing.com/>

Arts Numériques :

<http://arts-numeriques.net/spip.php?article215>

Hyperbate :

http://hyperbate.fr/dernier/?page_id=2482

Croyez Au Futur :

<https://sites.google.com/site/croyezaufutur/processing>

CreativeApplications.Net :

<http://www.creativeapplications.net/category/processing/>

Vimeo : <http://vimeo.com/tag:processing.org>

del.icio.us : <http://delicious.com/tag/processing>

Flickr : <http://www.flickr.com/groups/processing/>

Les livres <http://www.learningprocessing.com/>

<http://processing.org/shop/>

<http://www.complexification.net/>

1.4 Extensions et projets alternatifs

Les projets *Wiring*⁸ et *Arduino*⁹, qui se basent sur *Processing*, permettent la manipulation de circuits électroniques extérieurs pour interfacer des capteurs ou des appareils électroniques divers (servomoteurs, leds,...).

Un autre environnement logiciel, *Fritzing*¹⁰, a été conçu pour aider les utilisateurs au prototypage de matériel destiné à être utilisé avec Arduino et Processing. Le projet Mobile Processing¹¹, comme son nom l'indique, a l'ambition de porter Processing sur téléphones portables. Enfin, *Processing.js*¹² est une implémentation javascript de Processing.

Depuis sa version 1.5.1, Processing peut produire des applications pour plate-forme Android et accéder aux capteurs spécifiques aux dispositifs mobiles (accéléromètre, GPS, etc.). Dans la version alpha de Processing 2.0 (prévu pour fin 2011), Processing pourra aussi exporter directement du contenu au format HTML5/Javascript.

8. <http://wiring.org.co>

9. <http://www.arduino.cc/>

10. <http://fritzing.org/>

11. <http://mobile.processing.org/>

12. <http://processingjs.org/>

Installation de *Processing*

Sommaire

2.1 Sous Windows	5
2.2 Sous Mac OS X	5
2.3 Sous GNU/Linux	6

Processing étant écrit en Java, il fonctionne en principe sous toute plate-forme supportant au moins la version 5 (également appelée 1.5) de ce langage. Les plates-formes les mieux supportées sont néanmoins Microsoft Windows (Windows XP et Windows 7), GNU/Linux et Mac OS X. L'installation proprement dite de *Processing* est assez simple et dépend de la plate-forme que vous utilisez. Dans tous les cas, allez sur la page de téléchargement <http://processing.org/download/> et cliquez sur le nom de votre plate-forme. Pour la suite de ce chapitre, dans les noms de fichiers et dossiers, le xxx fait référence à la version de *Processing* utilisée.

2.1 Sous Windows

Pour Windows, il est préférable de ne pas utiliser l'option Without JAVA qui nécessitera d'installer séparément Java. En cliquant sur Windows dans la page de téléchargement du site de *Processing*, vous allez télécharger une archive processing-xxx.zip. Une fois le téléchargement achevé, décompressez l'archive et placez le dossier *Processing* extrait de l'archive dans le dossier C:\Program Files\. Allez ensuite dans le dossier C:\Program Files\Processing et exécutez le fichier processing.exe en cliquant dessus.

2.2 Sous Mac OS X

La version Mac OS X est téléchargée sous la forme d'une image disque (fichier .dmg). Par défaut ce fichier s'affiche sur le bureau et la fenêtre de l'image

disque s'ouvre automatiquement. Au sein de cette fenêtre, il suffit de prendre l'icône *Processing* et de la glisser sur l'icône du dossier Applications.

Pour exécuter le programme que vous venez d'installer, rendez-vous dans votre dossier Applications et double cliquez sur l'icône *Processing*.

2.3 Sous GNU/Linux

Après avoir cliqué sur Linux pour télécharger le fichier `processing-xxx.tgz` correspondant à cette plate-forme, il suffit de suivre pas à pas la procédure d'installation décrite ci-après.

Enregistrer le fichier sur votre ordinateur

Il est préférable d'enregistrer (ou copier) le fichier téléchargé dans votre répertoire personnel (exemple : `/home/MonDossierPersonnel`).

Extraire les contenus du fichier

Cette opération peut être réalisée de 2 manières : en utilisant votre souris (notamment si vous utilisez la distribution Ubuntu - gnome) : effectuez un clic droit sur le fichier pour faire apparaître le menu contextuel puis cliquez sur Extraire ici (Extract here). pour les plus experts, saisissez et exécutez dans un terminal la commande suivante :

```
tar -zxvf processing-xxx.tgz
```

Dans les deux cas, un dossier `processing-xxx` sera créé dans votre répertoire personnel, le `xxx` faisant référence version du logiciel. Ainsi dans le cas de la version 1.2.1 de *Processing* le nom du dossier s'appellera `processing-1.2.1` et le chemin pour accéder à ce répertoire sera `/home/MonDossierPersonnel/processing-1.2.1`.

Installer *Processing*

Pour installer *Processing*, en mode graphique, il suffit de se placer dans le dossier créé et d'effectuer un double clic. Si par contre vous utilisez un terminal, saisissez la commande : `cd /home/VotreDossierPersonnel/Processing-xxx/`

Autoriser l'exécution de *Processing*

Après l'installation du logiciel, il faut s'assurer que le fichier *Processing* contenu dans le dossier du programme (`/home/MonDossierPersonnel/Processing-xxx`) est bien exécutable. Ce n'est pas le cas par défaut. Dans un terminal, exécutez la commande suivante :

```
chmod +x processing
```

En mode graphique, effectuez un clic droit sur le fichier *Processing* puis dans le menu contextuel qui apparaît, cliquez sur propriétés. La boîte de dialogue suivante s'affichera :

Cliquez sur l'onglet Permissions puis cocher la case Autoriser l'exécution du fichier comme un programme.

Lancer *Processing*

Pour démarrer le programme *Processing* dans le terminal (en étant toujours dans le dossier du programme, exemple `/home/MonDossierPersonnel/processing-1.2.1/`), lancer la commande :

```
./processing
```

En mode graphique, faites un double clic sur le fichier *Processing* puis dans la boîte de dialogue qui apparaît, cliquez sur le bouton Lancer.

Installer Java, si besoin

Pour fonctionner, *Processing* requiert Java (au moins la version 5, également appelée 1.5) originaire de Sun Microsystems. Si Java n'est pas installé, vous pourrez télécharger et installer openjdk en vous rendant sur le site <http://openjdk.java.net/>, ou mieux, en utilisant le système de paquetage de votre distribution de GNU/Linux. Sous Ubuntu, exécuter la commande suivante (notez bien que cette étape est préalable à l'installation de JRE présentée un peu plus loin) :

```
sudo apt-get install openjdk-6-jdk
```

Ensuite, il se peut que vous ayez à changer la version de Java qui est proposée par défaut. Pour vous en assurer, saisissez la commande suivante :

```
sudo update-alternatives --config java
```

Une liste de choix devrait alors apparaître. Entrez le chiffre qui se trouve sur la même ligne que le chemin qui contient le mot `☐sun☐`, et appuyer sur entrée. Dans l'exemple présenté ci-dessous, la version présélectionnée est identifiée par un astérisque situé au début d'une des lignes. Il faudrait donc saisir le chiffre 4 pour sélectionner la bonne version.

Introduction au logiciel *Processing*

Sommaire

3.1 Les bases du langage	9
3.1.1 L'interface	9
3.1.2 Premières clés	10
3.2 L'espace de dessin	12

Processing propose à la fois un environnement de création complet et un ensemble de fonctionnalités supplémentaires qui viennent enrichir les possibilités du logiciel. Cet environnement permet d'écrire des programmes (appelés *sketchs* dans *Processing*), de les convertir en fichiers autonomes, de les publier ainsi que d'identifier et de corriger les erreurs. Il contient les fonctions essentielles à la programmation tout en étant simple d'utilisation.

3.1 Les bases du langage

Cette partie vous présente les bases de l'interface de *Processing* et les notions minimales de la syntaxe Java à connaître pour bien débiter. A ce jour, l'interface de *Processing* est disponible uniquement en anglais. Malgré cela,

son utilisation reste simple.

3.1.1 L'interface

L'interface d'utilisation de *Processing* est composée de deux fenêtres distinctes : la fenêtre principale dans laquelle vous allez créer votre projet et la fenêtre de visualisation dans laquelle vos créations (dessins, animations, vidéos) apparaissent.

On trouve plus précisément les éléments suivants dans l'interface :

1. Barre d'actions
2. Barre d'onglets
3. Zone d'édition (pour y saisir votre programme)

4. Console (destinée aux tests et messages d'erreur)
5. Fenêtre de visualisation (espace de dessin)
6. Barre de menu (non représentée sur la copie d'écran)

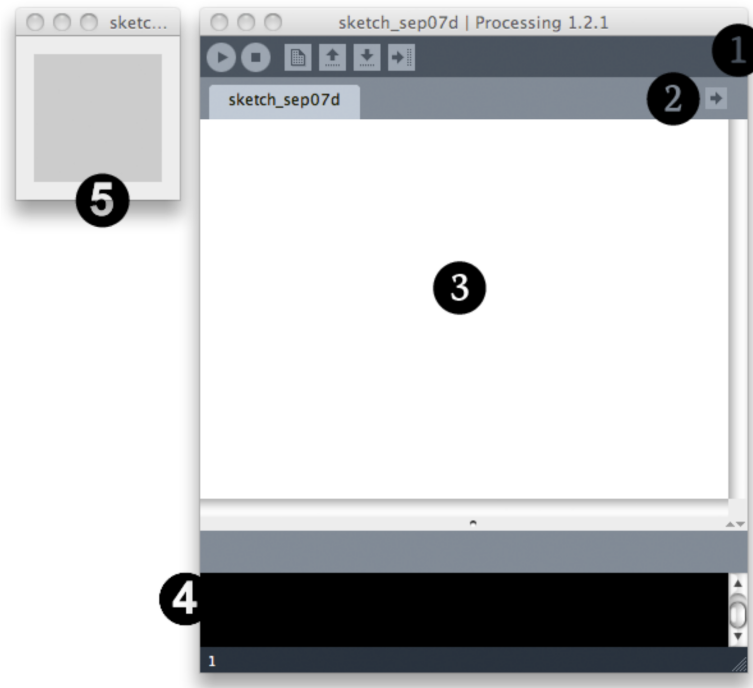


Figure 3.1: *Interface Processing*

Barre d'actions Dans l'ordre, de gauche à droite :

- Bouton "Run" : exécute votre sketch (votre programme).
- Bouton "Stop" : arrête l'exécution de votre sketch.
- Bouton "New" : Crée un nouveau sketch.

- Bouton "Open" : ouvre un sketch existant.
- Bouton "Save" : sauvegarde le sketch en cours.
- Bouton "Export" : exporte le sketch pour le web.

Le dossier de travail C'est le dossier dans lequel seront enregistrés les sketches et les bibliothèques (des modules externes proposant des fonctionnalités supplémentaires). Par défaut ce dossier se nomme *Processing* et se trouve dans Documents (sous Mac) ou Mes Documents (sous Windows). Sous GNU/Linux, il est dans votre dossier personnel sous le nom de sketchbook.

Pour modifier ce dossier, allez dans le menu *Files > Preferences*. Dans la boîte de dialogue qui apparaît, cliquez sur *Browse (parcourir)* pour choisir le dossier qui vous convient.

A tout moment, pour savoir quel est votre dossier de travail, sélectionnez, dans le menu *Sketch > Show Sketch Folder*.

3.1.2 Premières clés

Processing utilise le langage Java pour créer vos programmes. Ce langage, qui va être lu par votre ordinateur après avoir cliqué sur le bouton de lancement du sketch, possède un certain nombre de règles de syntaxe qui si elles ne sont pas respectées empêcheront l'exécution correcte du programme. Il y a aussi un certain nombre de concepts de base nécessaires à connaître.

Majuscules et minuscules *Processing* est sensible à la casse, il fait la différence entre les majuscules et les minuscules :

libre est différent de Libre !

Le point virgule A la fin de chaque instruction (dessiner un cercle, effectuer un calcul, etc.), vous devez mettre un `;` afin de signaler à l'ordinateur la fin de l'instruction. Dans l'exemple ci-dessous, on utilise les caractères `□ //`

□ pour insérer un commentaire qui sera ignoré lors de l'exécution (la présence de commentaires dans votre programme facilite sa compréhension ultérieure).

```
//Dessine un cercle
ellipse(10,10, 10, 10);
//Crée une variable
int chiffre = 10 + 23;
```

Appels de méthodes *Processing* propose un grand nombre de fonctionnalités prédéfinies appelées méthodes : dessiner un rectangle, définir une couleur, calculer une racine carrée, etc. Ces méthodes ont chacune un nom spécifique. Pour faire appel à elles, il suffit de taper leur nom en respectant les majuscules et minuscules et de coller des parenthèses après le nom : parfois on doit préciser certaines valeurs à l'intérieur des parenthèses (couleur, position, taille, etc.). L'exemple ci-dessous affiche un cercle gris.

```
fill(128);
ellipse(50, 50, 60, 60);
```

Affichage dans la console La console (la zone 4 dans le schéma de l'interface présenté en début de ce chapitre) permet d'afficher du texte brut à des fins de test et de débogage (correction d'erreurs). Pour afficher quelque chose dans cette zone, il faut utiliser la méthode `println()` ;

```
println("Salut tout le monde!");
println(1000);
```

Par extension, le terme console sert également à désigner la bande grise juste au dessus de la zone noire de test/débogage : *Processing* y affiche un certain nombre de messages, notamment pour signaler des erreurs.

Opérations arithmétiques *Processing* permet d'effectuer des calculs mathématiques. Tout au long de l'utilisation de cet environnement, vous serez amené à calculer des valeurs. Ne vous inquiétez pas : l'ordinateur le fera pour vous. Les opérations d'addition, soustraction, multiplication et division peuvent être combinées. Il est également possible d'utiliser des parenthèses pour définir l'ordre des opérations. Attention aux nombres à virgule ! Dans *Processing*, les unités sont séparées des décimales par un point et non par une virgule. Voici quelques exemples d'opérations arithmétiques :

```
println(10 + 5);
println(10 + 5 * 3); // 5*3 (soit 15) puis additionne 10
println((10 + 5) * 3); // 10+5 (soit 15) puis multiplie 15 par 3
println(10.4 + 9.2);
```

Cette suite d'instructions saisie dans la fenêtre d'édition de *Processing* va produire dans la console le résultat suivant : Certaines opérations arithmétiques peuvent être contractées. Par exemple, `i++` donne le même résultat que `i = i + 1`. Et `x+=10` donne le même résultat que `x=x+10`. Maintenant que vous connaissez les bases de l'interface de *Processing*, il ne vous reste plus qu'à apprendre à écrire du code qui fonctionne pour dessiner et créer ce que vous souhaitez. C'est l'objet des prochains chapitres.

3.2 L'espace de dessin

L'espace de dessin constitue l'espace de représentation proprement dit. Cette fenêtre de visualisation affichera vos réalisations dans *Processing* en 2 ou 3 dimensions. Cet espace est créé par l'instruction `size()` qui prend deux arguments : `size(largeur, hauteur)` ;. Par exemple, dans la fenêtre d'édition du logiciel *Processing*, saisissez la commande suivante :

```
size(323,200);
```

Puis cliquez sur le bouton **Run**, votre fenêtre de visualisation se crée. Amusez-vous à changer les dimensions de cette fenêtre en modifiant les valeurs entre parenthèses pour en voir le résultat. Par défaut : `size()` ;...affichera une fenêtre de 100 pixels sur 100 pixels. Félicitations, vous avez créé votre première fenêtre de visualisation !

Coordonnées dans l'espace

Quand on travaille en 2 dimensions (2D), on utilise deux axes de coordonnées x et y correspondant respectivement à la largeur (axe horizontal) et à la hauteur (axe vertical) d'une situation. Par convention de la mesure de

l'espace, le coin en haut à gauche correspond aux valeurs $x=0$ et $y=0$. Les valeurs x sont croissantes vers la droite et les valeurs y sont croissantes vers le bas, contrairement à notre habitude du plan cartésien. Ces valeurs x et y peuvent s'étendre théoriquement à l'infini, même si, en réalité, les contraintes de la taille de votre fenêtre vont délimiter la taille maximale d'une surface de création visible. C'est donc dans cet espace que nous allons dessiner.

Quand on travaille en 3 dimensions (3D), en plus des deux axes de coordonnées, on a un troisième axe de coordonnées z, exprimant la profondeur : Dans ce cas précis, on utilise la commande `size` avec un troisième paramètre indiquant que l'on travaille dans un espace en 3D :

```
size(100, 100, P3D);
```

Connaître la taille de l'espace de dessin

Au sein d'un programme, on peut connaître à tout moment la taille de l'espace de dessin utilisé au moyen des mots-clés `width` et `height`. Ces instructions sont très utiles lorsque l'on souhaite notamment dessiner des formes qui puissent s'adapter ultérieurement aux éventuels changements de dimension de la fenêtre de visualisation.

Dessiner avec *Processing*

Sommaire

4.1 Les formes	13
4.2 Les couleurs	19
4.3 Le texte	23
4.4 Les images	25
4.5 Les styles de bordures	32
4.6 La typographie	34
4.7 Les transformations	38

Logiciel de création multimédia, *Processing* possède la particularité d'utiliser des instructions informatiques pour dessiner, réaliser des animations en 2 ou 3 dimensions.

4.1 Les formes

Beaucoup de formes prédéfinies sont fournies par *Processing*. Nous en distinguons 2 types, les formes 2D et les formes 3D.

En 2D

Le point Pour commencer à dessiner, nous allons partir d'un point. Sur l'écran, un point est l'équivalent d'un pixel localisé dans la fenêtre de visualisation par deux axes de coordonnées, x et y correspondant respectivement à la largeur (axe horizontal) et à la hauteur (axe vertical) de l'espace de dessin. En suivant ce principe, la création d'un point dans *Processing* s'effectue à l'aide de l'instruction `point(x,y)`. Dans cet exemple, le point est très petit. Il est placé au centre de la fenêtre de visualisation.

```
point(50, 50);
```

Notez que le cadre de la fenêtre de visualisation (l'espace de dessin) a une dimension de 100x100, ce qui explique que le point soit situé en plein milieu. Si on le dessinait en dehors du cadre (hors champ), avec par exemple l'instruction `size(150,150)`, on ne le verrait pas.

La ligne Par définition, une ligne (AB) est constituée par une infinité de points entre un point de départ A et un point d'arrivée B. Pour la construire, nous allons nous intéresser uniquement aux coordonnées x et y de A et de B. Ainsi, si par exemple dans la fenêtre par défaut, le point A se situe dans la région en bas à gauche de votre fenêtre, et que le point B se situe en haut à droite, les instructions suivantes, peuvent dessiner cette ligne sous la forme `line(xA,yA,xB,yB)` :

```
line(15, 90, 95, 10);
```

Le rectangle Un rectangle se dessine par quatre valeurs en faisant l'appel de `rect(x,y,largeur,hauteur)`. La première paire de valeurs x et y, par défaut (mode CORNER) correspond au coin supérieur gauche du rectangle, à l'instar du point. En revanche la seconde paire de valeurs ne va pas se référer à la position du coin inférieur droit, mais à la largeur (sur l'axe des x, horizontal) et à la hauteur (sur l'axe des y, vertical) de ce rectangle. Exemple :

```
rect(10, 10, 80, 80);
```

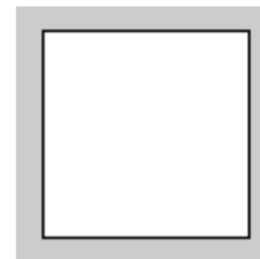


Figure 4.1: *Un rectangle.*

Comme les deux dernières valeurs (largeur et hauteur) sont identiques, on obtient un carré. Amusez-vous à changer les valeurs et observez-en les résultats. Pour que la première paire de valeurs corresponde au centre (le croisement des deux diagonales aux coordonnées 50, 50) du rectangle, il faut utiliser le mode CENTER comme suit :

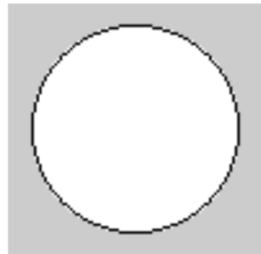
```
rectMode(CENTER);  
rect(50, 50, 80, 40);
```

Cela donne le résultat identique à l'exemple précédent.

L'ellipse Comme pour le rectangle, l'ellipse se construit sous les modes CENTER (par défaut), et CORNER. Ainsi l'instruction suivante produit un cercle dont les coordonnées du centre sont les deux premières valeurs entre parenthèses. La troisième valeur correspond à la grandeur du diamètre sur l'axe horizontal (x) et la quatrième à la grandeur du diamètre sur l'axe vertical : notez que si les 3e et 4e valeurs sont identiques, on dessine un cercle et dans le cas contraire, une ellipse quelconque :

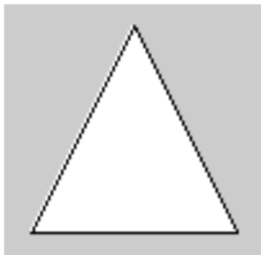
```
ellipse(50, 50, 80, 80);
```

Amusez-vous à faire varier les 3e et 4e valeurs et observez-en les résultats.

Figure 4.2: *Un cercle.*

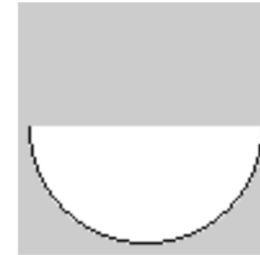
Le triangle Le triangle est un plan constitué de trois points. L'invocation de `triangle(x1,y1,x2,y2,x3,y3)` définit les trois points de ce triangle :

```
triangle(10, 90, 50, 10, 90, 90);
```

Figure 4.3: *Un triangle.*

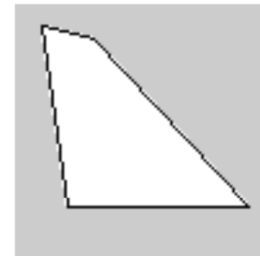
L'arc Un arc ou section de cercle, peut se dessiner avec l'appel de `arc(x, y, largeur, hauteur, début, fin)`, où la paire `x, y` définit le centre du cercle, la seconde paire ses dimensions et la troisième paire, le début et la fin de l'angle d'arc en radians :

```
arc(50, 50, 90, 90, 0, PI);
```

Figure 4.4: *Un arc.*

Le quadrilatère Le quadrilatère se construit en spécifiant quatre paires de coordonnées `x` et `y` sous la forme `quad(x1,y1,x2,y2,x3,y3,x4,y4)` dans le sens horaire :

```
quad(10, 10, 30, 15, 90, 80, 20, 80);
```

Figure 4.5: *Un quadrilatère.*

La courbe Une courbe se dessine à l'aide de `curve(x1, y1, x2, y2, x3, y3, x4, y4)`, où `x1` et `y1` définissent le premier point de contrôle, `x4` et `y4` le second point de contrôle, `x2` et `y2` le point de départ de la courbe et `x3`, `y3` le

point d'arrivée de la courbe :

```
curve(0, 300, 10, 60, 90, 60, 200, 100);
```



Figure 4.6: Une courbe.

La courbe de Bézier La courbe de type Bézier se construit à l'aide de `bezier(x1,y1,x2,y2,x3,y3,x4,y4)` :

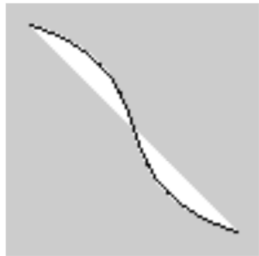


Figure 4.7: Une courbe de Bézier.

```
bezier(10, 10, 70, 30, 30, 70, 90, 90);
```

La courbe lissée L'appel de `curveVertex()` dessine plusieurs paires de coordonnées x et y, entre deux points de contrôle, sous la forme `curveVer-`

`tex(point de contrôle initial,xN,yN,xN,yN,xN,yN, point de contrôle final)` ce qui permet de construire des courbes lissées :



Figure 4.8: Une courbe de lissée.

```
beginShape();
curveVertex(0, 100);
curveVertex(10, 90);
curveVertex(25, 70);
curveVertex(50, 10);
curveVertex(75, 70);
curveVertex(90, 90);
curveVertex(100, 100);
endShape();
```

Formes libres Plusieurs formes libres peuvent être dessinés par une succession de points en utilisant la suite d'instructions `beginShape(),vertex(x,y),...,endShape()`. Chaque point se construit par ses coordonnées x et y. La fonction `CLOSE` dans `endShape(CLOSE)` indique que la figure sera fermée, c'est-à-dire que le dernier point sera relié au premier, comme dans l'exemple ci-dessous de dessin d'un hexagone :

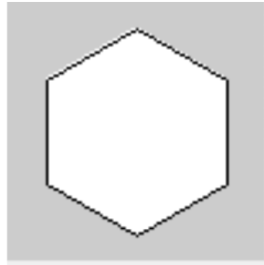


Figure 4.9: Une forme libre.

```
beginShape();  
vertex(50, 10);  
vertex(85, 30);  
vertex(85, 70);  
vertex(50, 90);  
vertex(15, 70);  
vertex(15, 30);  
endShape(CLOSE);
```

Contours & Remplissage Vous avez remarqué que jusqu'à présent, toutes les figures données en exemple comportent un contour, ainsi qu'une surface de remplissage. Si vous voulez rendre invisible le contour, utilisez `noStroke()` en faisant bien attention de le placer avant la forme à dessiner :

```
noStroke();  
quad(10, 10, 30, 15, 90, 80, 20, 80);
```

De la même manière, il est possible de dessiner des formes sans surface

de remplissage avec l'instruction `noFill()` :

```
noFill();  
quad(10, 10, 30, 15, 90, 80, 20, 80);
```

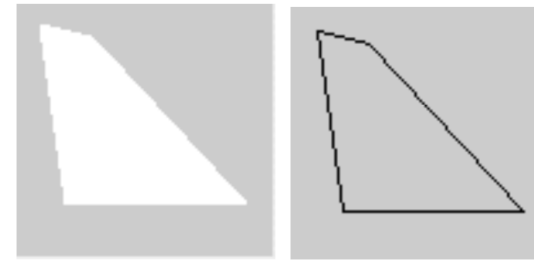


Figure 4.10: Pas de contours pour la forme tracée (gauche), pas de remplissage pour la forme tracée (droite).

Par défaut, l'arrière-fond de la fenêtre de visualisation (l'espace de dessin) est gris neutre, les contours des figures sont noirs, et la surface de remplissage est blanche. Vous apprendrez au prochain chapitre comment modifier les couleurs à votre convenance.

En 3D

Les formes prédéfinies disponibles en 3 dimensions (primitives 3D) peuvent être réalisées de manière simple en appelant `size(x, y, P3D)` au début de notre sketch puis en employant en fonction de vos besoins les instructions `sphere(taille)` et `box(longueur, largeur, profondeur)`. Il est également possible de produire des effets d'éclairage sur nos formes tridimensionnelles à l'aide de `lights()`.

La sphère La sphère peut se construire comme suit :

```
size(100, 100, P3D);  
noStroke();  
lights(); // éclairer l'objet 3D  
translate(50, 50, 0); // voir section "Transformations"  
sphere(28);
```



Figure 4.11: *Une sphère.*



Figure 4.12: *Une boîte.*

La boîte Voici un exemple de construction de boîte :

```
size(100, 100, P3D);  
noStroke();  
lights();  
translate(50, 50, 0);  
rotateY(0.5);  
box(40);
```

4.2 Les couleurs

Dessiner une image à l'écran, c'est changer la couleur des pixels. Les pixels sont des petites zones, le plus souvent carrées, qui possèdent une couleur. Chaque couleur se définit par trois canaux qui sont le rouge, le vert et le bleu. Une valeur de 100% de chacun de ces trois canaux donne du blanc. Une valeur de 0% de chacun de ces trois canaux donne du noir. Il s'agit de lumière, et non de peinture. Ainsi, plus la valeur de chaque canal sera importante, plus la couleur sera lumineuse. Par exemple, 100% de rouge, 80% de vert et 20% de bleu vont donner ensemble la couleur orange. La méthode `fill()` nous permet de spécifier la couleur des prochaines formes à dessiner. Chaque canal d'une couleur peut être donné sur une échelle de 0 à 255. Ainsi, 80% de 255 donne 204, et 20% de 255 donne 51.

La couleur de fond

On peut changer la couleur de fond en appelant la méthode `background()`. Attention : rajouter `background()` à la suite d'une composition déjà existante, l'effacerait !

```
background(0, 0, 0);
```



Figure 4.13: *La couleur de fond.*

La couleur de remplissage

A chaque fois que l'on dessine une forme, on le fait avec la couleur de remplissage qui est choisie à ce moment-là. On le fait en appelant la méthode `fill()` :



Figure 4.14: *L'appel à fill().*

```
noStroke();  
fill(255, 204, 51);  
rect(25, 25, 50, 50);
```

Processing nous offre différents formats pour exprimer la couleur. Si vous faites

de la programmation Web, vous connaissez sans doute le format hexadécimal. Selon ce procédé, la même couleur orange peut être obtenue en écrivant :

```
fill(#ffcc33);
rect(25, 25, 50, 50);
```

Par ailleurs, il est possible de spécifier la valeur du canal alpha de la couleur utilisée, c'est-à-dire son degré de transparence. Pour ce faire, on doit donc préciser quatre paramètres à la méthode `fill()`. Le quatrième argument est la valeur alpha.

```
noStroke();
fill(255, 204, 51); // orange
rect(25, 25, 50, 50);
fill(255, 255, 255, 127); // blanc semi-transparent
rect(35, 35, 50, 50);
```



Figure 4.15: L'appel à `fill()` en spécifiant la valeur alpha.

Si l'on souhaite choisir une couleur qui correspond à un ton de gris, il suffit de donner un seul paramètre à la méthode `fill()`. C'est la valeur de gris, de 0 à 255.

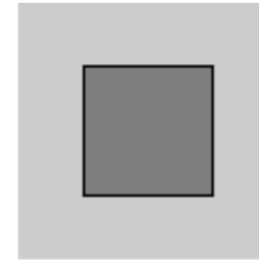


Figure 4.16: L'appel à `fill()` avec un seul paramètre.

```
fill(127);
rect(25, 25, 50, 50);
```

On peut désactiver le remplissage des formes en appelant la méthode `noFill()`.

La couleur du contour

Pour changer la couleur du contour des formes que l'on dessine, on doit appeler la méthode `stroke()` avec comme paramètres les canaux de la couleur désirée. Appeler `noStroke()` désactive la couleur de contour. A titre d'illustration, voici un dessin de masque africain utilisant ces deux instructions :



Figure 4.17: Un exemple utilisant stroke() et noStroke().

```
size(200, 200);
smooth();
background(255); // on dessine un fond blanc
stroke(#000000); // le contour sera noir
fill(#FFCC66); // le remplissage sera jaune
strokeWeight(3);

translate(width / 2, height / 2);

ellipse(0, 0, 100, 180); // forme elliptique du masque
ellipse(0, 60, 20, 25); // ellipse de la bouche

stroke(255, 0, 0); // le contour sera rouge
ellipse(28, -30, 25, 10); // ellipse de l'oeil droit
```

```
stroke(0, 0, 255); // le contour sera bleu
ellipse(-27, -30, 25, 10); // ellipse de l'oeil gauche

noFill(); // pas de remplissage pour la suite
stroke(#000000); // contour noir pour la suite
bezier(-30, -70, -5, -60, -5, 0, -5, 20); // sourcil droit
bezier(30, -70, 5, -60, 5, 0, 5, 20); // sourcil gauche
line(-5, 20, 5, 20); // base du nez
```

La portée des modifications de couleur

Par défaut, toute modification de style (couleur de remplissage ou de contour, épaisseur ou forme de trait) s'appliquera à tout ce que vous dessinerez ensuite. Pour limiter la portée de ces modifications, vous pouvez les encadrer par les commandes pushStyle() et popStyle().

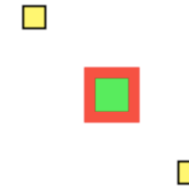


Figure 4.18: Les commandes pushStyle() et popStyle() pour limiter la portée des modifications de style.

```
size(100, 100);
background(255);
```

```
stroke(#000000);
fill(#FFFF00);
strokeWeight(1);
rect(10, 10, 10, 10);
```

```
pushStyle(); // On ouvre "une parenthèse" de style
```

```
stroke(#FF0000);
fill(#00FF00);
strokeWeight(5);
rect(40, 40, 20, 20);
```

```
popStyle(); // On ferme notre parenthèse de style
```

```
rect(80, 80, 10, 10);
```

Supprimez les commandes `pushStyle()` et `popStyle()` et observez la différence de comportement.

L'espace colorimétrique

Définir des couleurs à partir des canaux rouge, vert et bleu constitue un moyen parmi d'autres de décrire l'espace colorimétrique de votre dessin. *Processing* supporte également le mode TSV. TSV signifie *teinte, saturation, valeur*. En anglais, on appelle ce mode HSB, pour *hue, saturation, brightness*. On choisit une échelle de 0 à 100 pour chacun de ces 3 canaux. La teinte correspond à un chiffre indiquant la position de la couleur sur l'échelle chromatique, soit un arc-en-ciel. Le rouge est à gauche, puis vient l'orange, le jaune, le vert, le bleu et le violet. La méthode `colorMode()` est utile pour changer l'échelle numérique que l'on utilise pour spécifier les couleurs, et pour changer d'espace colorimétrique. Par exemple, appeler `colorMode(RGB, 1.0)`, va

changer l'échelle que l'on utilise pour spécifier chaque canal des couleurs afin qu'il aille de zéro à un. Ici, on change le mode de couleur pour le TSV afin de créer un dégradé de teinte qui ressemble aux couleurs d'un arc-en-ciel.



Figure 4.19: L'espace colorimétrique : le mode TSV.

```
noStroke();
size(400, 128);
// La teinte sera spécifiée avec un chiffre de 0 à 400
colorMode(HSB, 400, 100, 100);
// Quatre cent répétitions, voir section "Les répétitions"
for (int i = 0; i < 400; i++) {
  fill(i, 128, 128);
  rect(i, 0, 1, 128);
}
```

4.3 Le texte

On va maintenant dessiner des caractères textuels qui s'afficheront dans la fenêtre de visualisation. Attention : écrire dans l'espace de dessin ne ressemble pas à l'écriture dans le sens classique du terme, notamment comme dans un traitement de texte. Le texte dans *Processing* ressemblerait plutôt à du *graffiti urbain*, c'est-à-dire à une sorte de peinture de caractères alphanumériques qui finiront tous par s'empiler les uns sur les autres. C'est davantage un procédé *typo-graphique* qu'un travail d'écriture, avec une insistance sur l'aspect graphique des mots ou des lettres que nous allons dessiner.

Bonjour Processing !

Tout comme les formes prédéfinies, telles les lignes ou les ellipses, il suffit d'une seule instruction dans notre sketch pour dessiner du texte. Ouvrez une nouvelle fenêtre d'édition, tapez la ligne suivante, et exécutez-la en cliquant sur le bouton run :

```
text("Salut!", 10, 20);
```

Vous devriez voir s'afficher la petite fenêtre de visualisation par défaut de 100 x 100 pixels, avec le mot *Salut!* écrit en blanc.

Comme son nom l'indique, l'instruction `text()` dessine du texte dans la fenêtre de visualisation. Elle requiert trois paramètres : 1)le message que nous voulons écrire, 2)sa coordonnée x, 3)sa coordonnée y. Ce texte peut également être coloré, comme n'importe quelle forme géométrique, en changeant la couleur de remplissage :

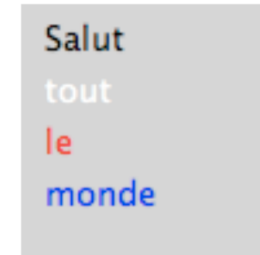


Figure 4.20: Un texte coloré.

```
fill(0);  
text("Salut", 10, 20);  
fill(255);  
text("tout", 10, 40);  
fill(255,0,0);  
text("le", 10, 60);  
fill(0,0,255);  
text("monde", 10, 80);
```

Ce programme devrait vous donner le résultat suivant, avec chacun des quatre mots écrits avec leur propre couleur. Il suffit d'indiquer une couleur, puis dessiner du texte avec cette couleur. Bien sûr vous pouvez ne choisir qu'une seule couleur et dessiner plusieurs messages avec cette couleur, ou dessiner avec une combinaison de couleurs comme dans cet exemple.

Empilement

Comme nous l'avons indiqué dans l'introduction de ce chapitre, les mots s'inscrivent à l'intérieur de la fenêtre de visualisation tel un dessin et non pas tel un système de traitement de texte. On ne peut ni sélectionner ni modifier

directement ce texte à l'aide de la souris ou du clavier. La seule modification qui peut être appliquée s'effectue au niveau de votre programme en précisant à l'aide d'instructions appropriées que vous souhaitez par la suite effacer complètement le texte et d'écrire par dessus. Voici une illustration rapide de ce principe de superposition :

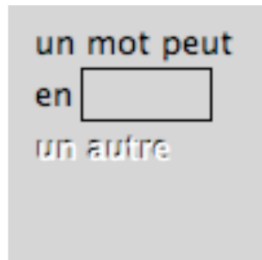


Figure 4.21: *Empilement de commandes.*

```
fill(0);  
text("un mot peut", 10, 20);  
text("en cacher", 10, 40);  
fill(204);  
rect(28,25,50,20);  
fill(0);  
text("un autre", 10, 60);  
fill(255, 255, 255);  
text("un autre", 11, 61);
```

Nous avons écrit quatre messages, parfois avec la même couleur, parfois avec une autre. Notez que le mot "cacher" a été caché justement par la couleur de remplissage qui a été définie avec la même valeur que celle utilisée pour dessiner le fond. Comme dans n'importe quel dessin, l'ordre des opérations

est important pour déterminer le rendu final. Ce principe s'applique même en dehors du logiciel *Processing* : "s'habiller, sortir, aller au travail" ne donnera pas le même résultat que "sortir, aller au travail, s'habiller".

4.4 Les images

Ce que nous appelons "image" dans *Processing* n'est en fait rien d'autre qu'une collection de pixels, rassemblés à l'intérieur d'un rectangle. Pour dessiner une image à l'écran, nous devons donner une couleur à chacun des pixels d'un rectangle, puis donner la position en x,y où nous voulons dessiner cette collection de pixels. Il est aussi possible de modifier la taille largeur,hauteur de notre image, même si ces dimensions ne correspondent pas à la taille originelle de l'image.

Trouver une image

Pour dessiner une image dans *Processing*, il faut commencer par trouver une image et l'importer dans notre sketch. Vous pouvez prendre une photo à l'aide de votre appareil numérique ou directement depuis votre webcam ou bien encore effectuer une recherche d'images se trouvant déjà dans le disque dur de votre ordinateur. Pour cet exercice, l'origine de l'image importe peu. Par contre, nous recommandons de commencer avec une image relativement petite, par exemple d'une largeur de 400 x 300 pixels. Ici nous allons commencer avec une image légèrement réduite de l'île des peupliers à Ermenonville, trouvée sur le site Commons de *Wikimedia* (base de données d'images et de médias appartenant au domaine public ou sous licence libre) à l'adresse suivante : <http://fr.wikipedia.org/wiki/Fichier:Erm6.JPG>

Formats d'image

Trois formats de fichier d'image sont acceptés dans *Processing* : PNG, JPEG, ou GIF. Ceux qui ont de l'expérience dans la conception de sites web devraient reconnaître ces trois formats, car ce sont les plus répandus sur la toile. Chaque

format utilise ses propres méthodes de compression de l'image (réduction de la taille mémoire de l'image occupée sur l'ordinateur), lui donnant à la fois des avantages et des désavantages et que nous pouvons résumer de la manière suivante :

1. JPEG est souvent utilisé pour compresser des images de type photographique. Ce format ne permet pas d'avoir des zones transparentes.
2. GIF est historiquement utilisé pour l'illustration de boutons et autres éléments graphiques de l'espace de travail d'un programme. Sur les sites internet, ce format est utilisé pour les logos et de manière générale pour les dessins réalisés par ordinateur (notamment ceux qui comportent des aplats de couleurs). Ce format peut contenir des zones transparentes binaires (soit opaques soit transparentes, sans possibilité d'opacité intermédiaire). Il existe des images gif animées, mais *Processing* ignorera cet aspect.
3. PNG est de plus en plus utilisé pour les deux usages (photos + dessins) et peut contenir des zones transparentes non binaires, offrant des niveaux d'opacité (opaque, semi-opaque, totalement transparent).

Glisser-déposer

Nous allons maintenant importer le fichier de cette image dans l'environnement *Processing*. Pour bien réussir cette étape, nous vous recommandons de sauvegarder d'abord votre sketch, de préférence dans le dossier *processing* qui devrait se trouver par défaut dans le dossier Documents ou Mes documents de votre ordinateur. Localisez maintenant votre fichier d'image, et glissez-le directement sur la fenêtre *Processing* : Lorsque nous glissons les fichiers de cette manière, *Processing* nous indique dans la bande grisée de la console qu'il a ajouté ce fichier "dans le sketch". En réalité, *Processing* a simple-

ment copié cette image dans un dossier nommé *data*, qui devrait maintenant se trouver à côté de votre *programme.pde* que vous venez de sauvegarder. C'est dans ce dossier *data* que nous devons placer en réalité toutes les images dont nous voulons nous servir dans notre sketch. C'est dans cet emplacement que doivent également être rangés les autres fichiers médias comme les polices ou les sons. Pour accéder rapidement à ce dossier, afin par exemple d'y placer de nouveaux fichiers d'images, il suffit de sélectionner, dans le menu *Sketch*, l'option *Show Sketch Folder*. Cette option est également accessible de sélectionner, dans le menu *Sketch*, l'option *Show Sketch Folder*. Cette option est également accessible via le raccourci *Ctrl-k* sur Windows et GNU/Linux ou *Cmd-k* sur Mac.

Importer une image

Maintenant que nous avons placé une image dans notre dossier *data*, nous pouvons désormais nous en servir dans notre programme.

```
size(500,400);  
PImage ile;  
ile = loadImage("ile.jpg");  
image(ile,50,10);
```

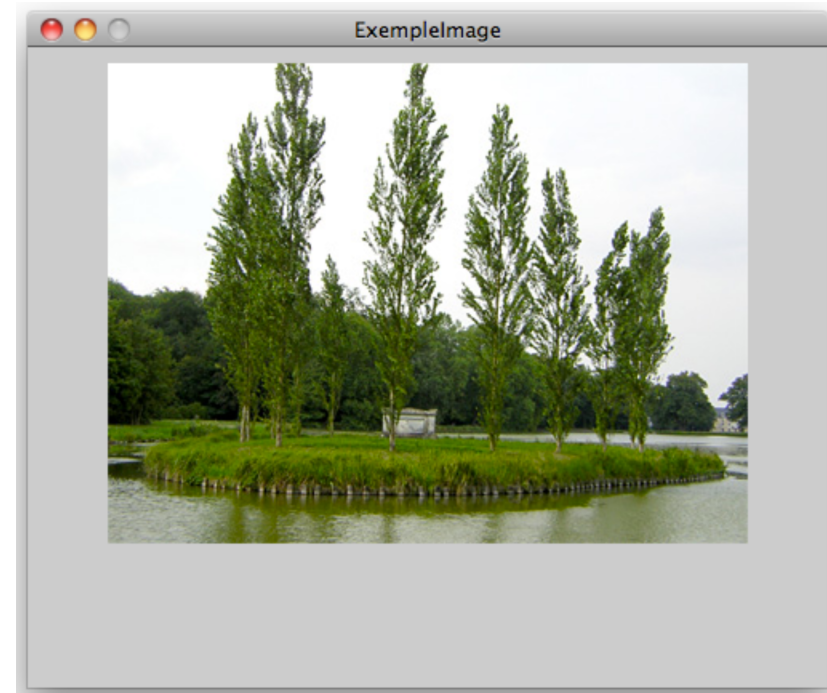


Figure 4.22: Exemple de programme chargeant l'image "ile.jpg".

D'abord nous avons donné à notre espace de dessin une taille plus grande que notre image, juste pour mieux comprendre comment celle-ci peut être positionnée dans le sketch final. Il y a trois étapes pour afficher une image dans *Processing* :

1. Créer une variable qui contiendra les données (en pixels) de notre image.
2. Importer les pixels d'un fichier dans notre variable.
3. Dessiner les pixels de cette variable dans notre espace de dessin.

Tout d'abord, nous devons créer une variable *Processing*, avant d'importer notre fichier dedans. Mais avant tout, à quoi sert une variable (voir section "Les variables") ? Et bien dans ce cas précis, il s'agit d'un nom interne à notre programme qui contient l'ensemble des pixels de notre fichier *ile.jpg*. A chaque fois que nous écrirons par la suite le mot "ile" dans notre programme, *Processing* comprendra qu'il s'agit de la suite de valeurs en pixels qui composent notre image. C'est justement le rôle de cette action `loadImage("nomdefichier")` d'aller chercher ces pixels dans le fichier et les importer dans notre variable nommé "ile". Vous avez peut-être également noté un mot étrange au tout début du code ci-dessus, le mot `PImage`. Celui-ci indique à *Processing* le genre de la variable et lui permettra de consacrer assez de mémoire dans votre ordinateur pour contenir l'ensemble des données de ce genre. Dans le cas de n'importe quelle variable, la syntaxe à utiliser est `type de la variable nom de la variable = les valeurs de la variable`. Par exemple, si par un tour de force il était possible d'importer un petit chien tout mignon dans *Processing*, il suffirait d'écrire `PetitChiot milou = loadDog("milou.dog")` ; On écrit d'abord le type de la chose, le nom de la chose, et enfin on lui donne sa valeur. Ici, cette valeur est donnée par la fonction `loadImage()` qui va aller chercher les pixels de l'image dans le fichier et les importera dans notre variable nommée *ile*. Pour plus d'informations sur les variables et les différents types de variables, reportez-vous à la section "Les variables". Enfin, une fois notre variable remplie, nous la dessinons à une position *x,y* dans notre sketch. Si nous voulons dessiner notre image à sa taille originelle, nous utilisons la version courte de l'instruction `image` qui nécessite uniquement trois paramètres `PImage, x, y` :

```
image(ile,50,10);
```

Importer une image web

Nous pouvons également importer des images directement depuis Internet, en indiquant l'adresse web de l'image à la place du nom de fichier.

```
size(400,400);  
PImage webcam;  
webcam = loadImage("http ://www.gutenberg.org/files/3913/3913-  
h/images/rousseau.jpg");  
image(webcam,10,20,width,height);
```

Si notre programme n'est pas animé (voir section "La méthode `draw`"), comme c'est le cas ici, il y aura juste une longue pause lorsque l'image se charge dans *Processing*. Par contre, faites bien attention à placer ce type d'instructions au début du sketch sinon vous risquez de ralentir fortement le fonctionnement des autres parties du programme, celles-ci devant attendre le téléchargement complet de vos images depuis Internet avant de s'exécuter. Dans des cas très spécifiques, si vous avez besoin d'importer des images web en plein milieu de votre sketch, sachez qu'il existe des techniques appropriées dénommées "fils d'exécution". Ces techniques séparent le chargement de fichiers médias depuis internet du reste des fonctions du programme. Il s'agit malheureusement d'un sujet trop avancé pour ce manuel. Pour davantage d'informations sur les fils d'exécution, reportez-vous au forum du site de *Processing* en faisant une recherche sur le mot "thread".

Changer de taille

En ajoutant deux paramètres, nous pouvons changer la taille de l'image. Cette taille peut être plus petite ou plus grande que l'image d'origine, et a priori il n'y a pas vraiment de limite. Par contre, au-delà de la taille d'origine

de l'image, *Processing* sera obligé d'inventer des pixels en doublant les originaux, ce qui donnera un effet pixelisé. Cet effet n'est pas forcément indésirable, à vous de voir. Pour changer la taille de l'image, il suffit de rajouter deux paramètres à votre image, *largeur*, *hauteur*, ce qui nous amène à 5 paramètres : *variableImage*, *x*, *y*, *largeur*, *hauteur*.

```
size(500,250);  
PImage ile;  
ile = loadImage("ile.jpg");  
image(ile,10,10,20,15);  
image(ile,20,20,50,30);  
image(ile,20,20,50,30);  
image(ile,45,45,100,75);  
image(ile,95,95,1000,750);
```

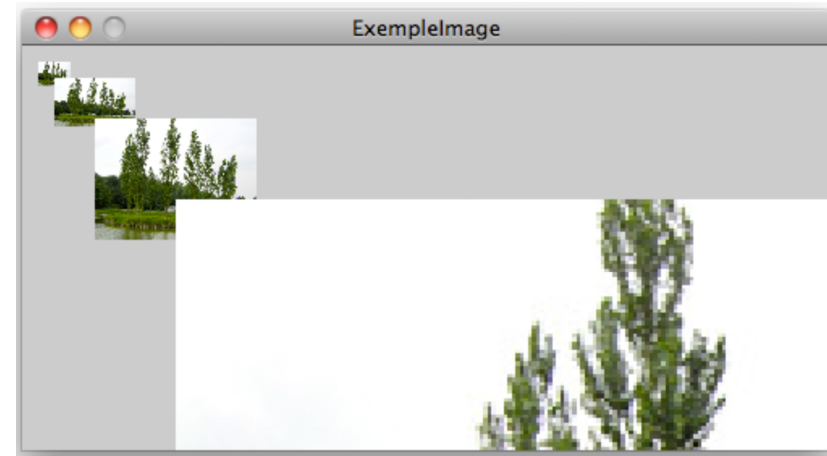


Figure 4.23: Exemple de programme chargeant l'image "ile.jpg" sous différentes tailles.

Notez que nous avons importé qu'une seule fois l'image dans notre variable *ile* et que celle-ci peut être dorénavant utilisée pour le restant de notre programme. Notez également que nous avons respecté les proportions *x,y* de notre image dans le changement de la taille, mais que celles-ci auraient pu prendre n'importe quelle valeur, ce qui reviendrait à étirer l'image sur un de ces deux axes.

Rendre transparent l'arrière-plan d'une image

Souvent nous avons besoin d'importer des images qui ne sont ni carrées, ni rectangulaires, comme dans le cas d'un petit jeu vidéo utilisant des silhouettes de personnages en deux dimensions : nous ne voulons pas voir en permanence un carré blanc autour du profil de notre héros. Malheureusement, à ce jour, les images à base de pixels doivent être importées dans un format carré. Pour

lever cette contrainte, certains formats de fichiers font appel à une couche alpha pour gérer la transparence de certaines zones de l'image et ainsi laisser apparaître les éléments qui sont situés derrière elles (un fond coloré ou illustré par exemple). Cette couche se superpose aux couches rouge, vert, bleu utilisées pour composer l'image et indique l'intensité de la transparence pour chaque pixel, avec même la possibilité de pixels semi-transparents dans le cas du format PNG. Si vous voulez sauvegarder votre image avec une couche alpha, vous avez trois possibilités qu'il faut choisir lors de l'exportation de votre image dans votre logiciel de traitement d'image préféré. Chaque logiciel exportera les images à sa manière. Sachez juste qu'il existe une différence par exemple entre GIF, PNG-8, et PNG-24, notamment sur la façon dont chaque format va traiter cette couche alpha. Le plus sûr des trois, et qui vous donnera plus d'options, c'est le PNG-24.

parent.

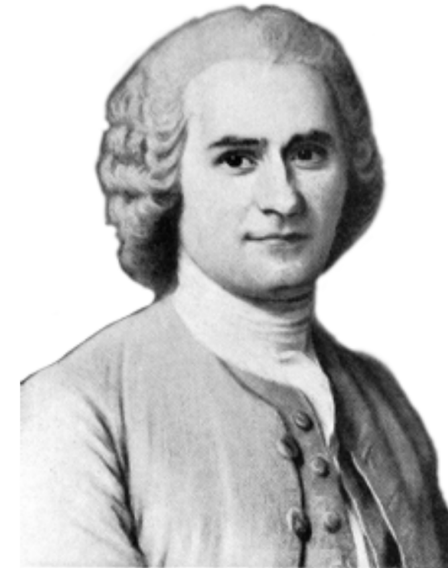


Figure 4.24: *Jean-Jacques Rousseau.*

Quel que soit le fond sur lequel elle sera placée, cette image s'y intégrera parfaitement parce qu'elle contient justement une couche alpha décrivant les parties transparentes et non transparentes du carré de pixels qui la composent. Notez que les pixels autour de Rousseau sont transparents, alors que son front et son cou sont opaques. N'hésitez pas à copier cette image au format png sur votre ordinateur depuis la version en ligne de ce manuel : <http://fr.flossmanuals.net/processing/>, chapitre Les images (conçue à partir d'une illustration mise à disposition sur le site Commons de Wikimedia, cette image est libre de droit). Dans notre programme *Processing*, nous allons

Voici une image du philosophe Jean-Jacques Rousseau avec un fond trans-

pouvoir constater que les pixels situés autour du personnage sont effectivement transparents en superposant l'image de Rousseau (au format png) avec celle du paysage précédemment utilisée (au format jpg) :

```
size(400,300);  
PImage ile;  
ile = loadImage("ile.jpg");  
PImage rousseau;  
rousseau = loadImage("rousseau.png");  
image(ile,0,0);  
image(rousseau,20,20);
```

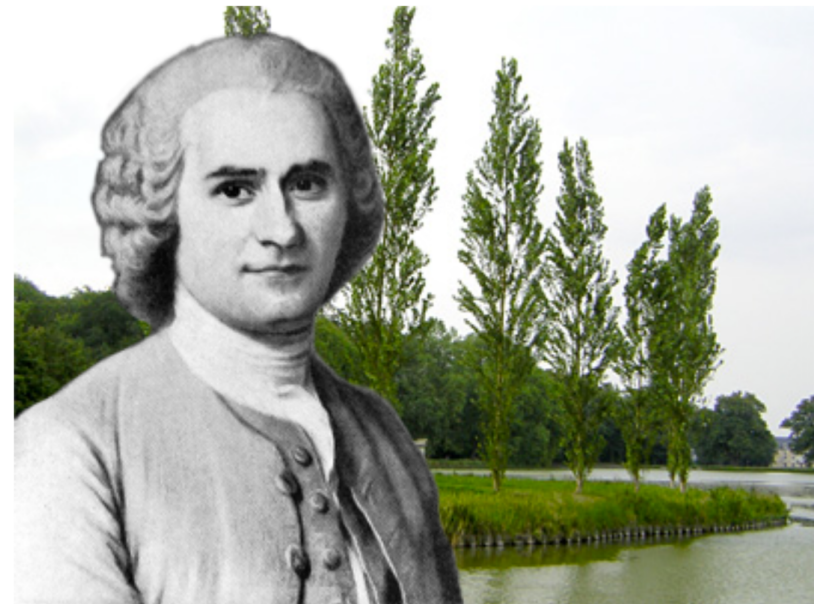


Figure 4.25: *Jean-Jacques Rousseau sur fond.*

Colorier les images

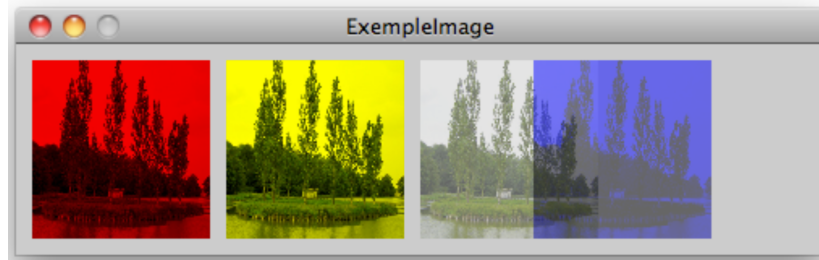


Figure 4.26: Image "ile.jpg" avec différentes teintes.

On peut également colorier les images. Autrement dit, on peut changer la couleur des pixels, soit dans leur ensemble, soit individuellement. La plus simple de ces méthodes concerne le coloriage de l'ensemble des pixels. Cette méthode de coloriage ressemble à peu près au coloriage de rectangles, mais nécessite une nouvelle instruction, `tint()`, pour le distinguer du coloriage direct des pixels à l'intérieur du rectangle. Dans le cas de `tint()`, *Processing* va modifier la couleur de chaque pixel au moment où celui-ci est dessiné dans l'espace de dessin.

```
size(500,130);
PImage ile;
ile = loadImage("ile.jpg");
tint(255,0,0);
image(ile, 10,10, 110,110);
tint(255,255,0);
image(ile, 130,10, 110,110);
tint(255,255,255,127);
image(ile, 250,10, 110,110);
```

```
tint(0,0,255,127);
image(ile, 320,10, 110,110);
```

Tout comme les instructions `fill()` et `stroke()`, l'instruction `tint()` peut prendre une, deux, trois, ou quatre valeurs, selon ce que nous voulons faire. En indiquant trois paramètres, par exemple, nous pouvons augmenter/diminuer l'intensité de la couche rouge, vert, ou bleu de notre image. En indiquant quatre paramètres, nous pouvons augmenter/diminuer, en plus de ces trois couleurs, la valeur de transparence/opacité de l'image.

4.5 Les styles de bordures

Le style des traits et des bordures des formes géométriques peut être ajusté afin d'éviter l'apparition d'effets graphiques indésirables sur les lignes obliques, aux intersections ou en bout de ligne. Différentes commandes permettant d'affiner le rendu sont présentées ci-dessous.

smooth

La méthode `smooth()` permet d'activer le lissage des contours. Elle permet d'éviter l'effet d'escalier qui apparaît sur les lignes diagonales.

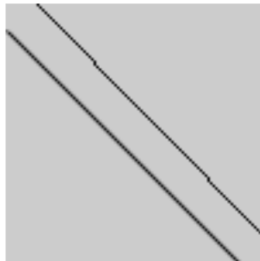


Figure 4.27: Le lissage des contours avec `smooth()`.

```
line(10, 0, 100, 90); // Ligne sans lissage
line(10, 0, 100, 90); // Ligne sans lissage
//On active le lissage
smooth();
line(0, 10, 90, 100); // Ligne lissée
```

strokeWeight

La méthode `strokeWeight()` permet de varier l'épaisseur d'une ligne ou d'un contour.

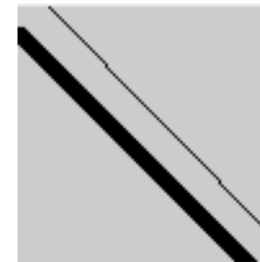


Figure 4.28: L'épaisseur des contours avec `strokeWeight()`.

```
line(10, 0, 100, 90); // Ligne de 1 pixel d'épaisseur
strokeWeight(5); //On définit l'épaisseur à 5 pixels
line(0, 10, 90, 100); // Ligne de 5 pixels d'épaisseur
```

strokeCap

La méthode `strokeCap()` permet de définir l'apparence des extrémités d'une ligne. Cette méthode n'est pas utile pour les formes. Elle peut avoir les valeurs `SQUARE` (extrémité carré), `PROJECT` (extrémité avec 2 petits angles brisés) ou `ROUND` (extrémité arrondie). Par défaut c'est le mode `ROUND` qui est utilisé. Cette méthode ne fonctionne pas avec P3D ou OpenGL.



Figure 4.29: L'apparence des extrémités des contours avec `strokeCap()`.

```
strokeWeight(10); // l'épaisseur des traits à 10 pixels
strokeCap(ROUND); // extrémité arrondie line(20, 40, 60, 80);
strokeCap(PROJECT); // extrémité avec 2 petits angles brisés
line(20, 20, 80, 80);
strokeCap(SQUARE); // extrémité carré
line(40, 20, 80, 60);
```

strokeJoin

La méthode `strokeJoin()` permet de modifier l'aspect des jointures. Elle peut avoir les valeurs `MITER`, `BEVEL` ou `ROUND`. Par défaut c'est le mode `MITER` qui est utilisé. Cette méthode ne fonctionne pas avec `P3D` ou `OpenGL`.

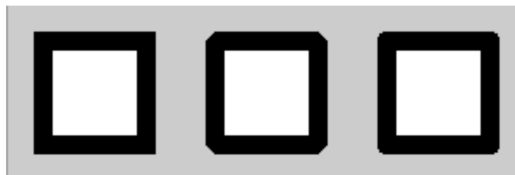


Figure 4.30: L'aspect des jointures des contours avec `strokeJoin()`.

```
size(300, 100); // modification de la taille du sketch
strokeWeight(10); // l'épaisseur des traits à 10 pixels
strokeJoin(MITER); // jointure carré rect(20, 20, 60, 60);
strokeJoin(BEVEL); // jointure brisée rect(120, 20, 60, 60);
strokeJoin(ROUND); // jointure arrondie rect(220, 20, 60, 60);
```

4.6 La typographie

Cette section va vous permettre de personnaliser l'usage des textes dans *Processing* en utilisant des polices de caractères alternatives.

La forme des mots

Si nous voulons dessiner avec une autre forme typographique que celle définie par défaut, il faut effectuer quelques étapes préalables :

1. Convertir une police de caractères en un format de fichier compatible avec *Processing* ;
2. Importer ce fichier dans le code du programme (ce fichier comporte toutes les informations graphiques décrivant l'apparence de la police utilisée) ;
3. Sélectionner cette police et l'activer dans notre programme ;
4. Dessiner du texte dans notre sketch à l'aide des instructions appropriées pour que le résultat s'affiche dans la fenêtre de visualisation de *Processing*.

Importer une police de caractères

Pour dessiner du texte dans notre fenêtre de visualisation, il faut choisir tout d'abord son apparence, en indiquant sa police de caractères. Pour bien réussir cette étape, nous vous recommandons de sauvegarder d'abord votre sketch dans votre dossier de travail (voir chapitre Bases de *Processing*). Une fois notre sketch sauvegardé, nous allons sélectionner, dans le menu Tools, l'action Create Font...

A priori nous devrions maintenant voir une fenêtre Create Font qui permet de convertir quasiment n'importe quelle police de caractère en une forme

utilisable dans notre sketch. Cette police doit être installée au préalable dans notre ordinateur pour qu'elle apparaisse dans cette liste.

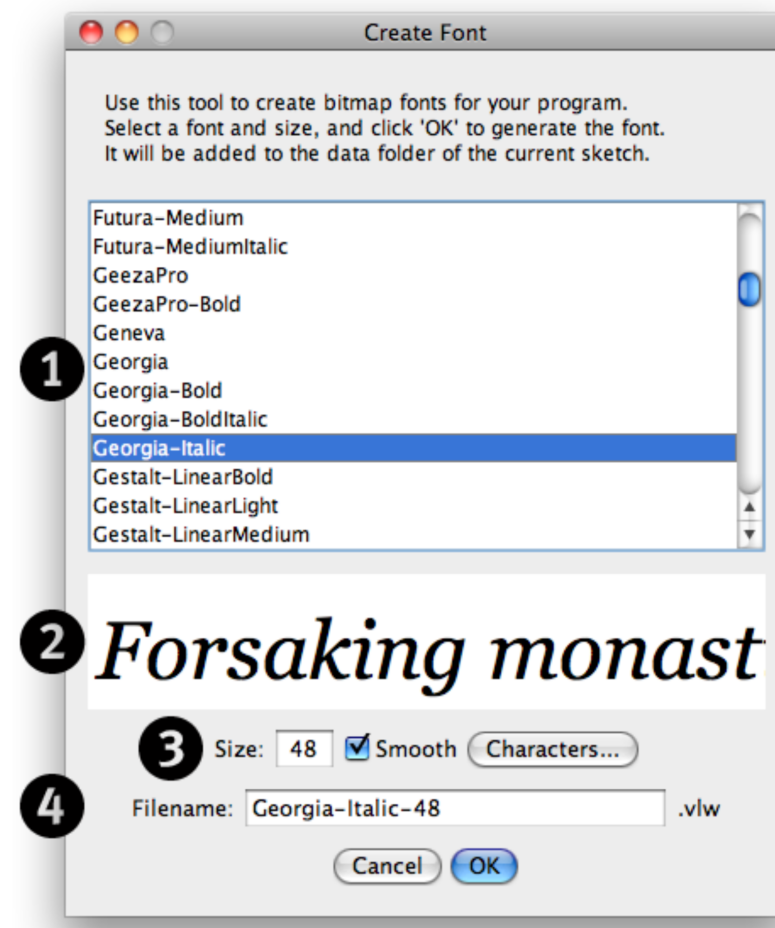


Figure 4.31: Fenêtre Create Font.

Cette fenêtre est décomposée en quatre parties :

1. La liste des polices actuellement installées sur notre ordinateur,
2. Une prévisualisation de la police actuellement sélectionnée, affichée à la taille indiquée dans le prochain champ (la zone numérotée 3 dans la copie d'écran ci-dessus),
3. A peu près la taille maximale à laquelle nous voulons dessiner cette police,
4. Le nom du fichier de cette police, une fois convertie dans le format natif de *Processing* (.vlw).

Vous avez peut-être noté qu'il existe également une case à cocher Smooth qui active/désactive l'anti-aliasing (fonction de lissage des polices pour éviter un effet de crénelage), ainsi qu'un bouton Characters... qui permet de préciser les caractères spéciaux qui doivent être inclus lors de la conversion de la police. Pour ne pas compliquer les choses, nous allons laisser ces deux options avec leurs valeurs par défaut.

Dans l'illustration ci-dessus nous avons sélectionné la police Georgia. C'est à partir du nom de cette police et de sa taille que *Processing* générera le fichier de la police à importer, ex : "Georgia-Italic-48.vlw". Notons enfin que l'extension ".vlw" associée à l'intitulé du fichier sera rajoutée à toutes les polices que nous importerons de cette manière. Si par curiosité vous vous intéressez à l'origine de cette extension, son nom fait référence sous forme d'un acronyme au "Visual Language Workshop" (vlw) du MIT Media Lab. C'est ce laboratoire qui historiquement est à l'origine d'un certain nombre de principes et de travaux qui ont permis à *Processing* de voir le jour. Si nous voulons savoir où *Processing* a sauvegardé notre police, il suffit de sélectionner, dans le menu Sketch, l'action *Show Sketch Folder*. Cette action fera apparaître le dossier "data" dans lequel notre police a été sauvegardée. Son fichier s'appelle

"Georgia-Italic-48.vlw". C'est ce nom que nous devons retenir pour intégrer la police dans notre programme.

Dessiner une phrase

Nous allons enfin dessiner avec notre police. Pour cela, il faut faire trois choses :

1. Importer le fichier Georgia-Italic-48.vlw dans une variable afin que notre programme connaisse le nom de la police utilisée et sache la dessiner lorsqu'il affichera du texte. Ce fichier contient en effet les informations décrivant la structure géométrique de la police pour pouvoir la reproduire ;
2. Sélectionner cette variable dans notre programme comme police active ;
3. Dessiner un caractère ou une phrase quelque part dans notre sketch à l'aide des instructions appropriées pour le voir ensuite s'afficher dans la fenêtre de visualisation de *Processing*.

En option, il est possible de choisir la taille à laquelle nous voulons dessiner avec notre police, mais comme nous avons déjà paramétré cet aspect lors de la création du fichier, il ne sera pas nécessaire de l'indiquer ici. Voici le code complet d'un programme simple qui réunit toutes ces étapes pour dessiner une phrase dans la fenêtre de visualisation. Par tradition, nous allons faire nos premiers pas dans l'écriture en écrivant "Salut tout le monde !".

```
size(500,150);
PFont police;
police = loadFont("Georgia-Italic-48.vlw");
textFont(police,48);
```

```
text("Salut tout le monde!", 20, 75);
```

Tout d'abord, nous avons fixé la taille de notre fenêtre de visualisation (l'espace de dessin), comme dans quasiment n'importe quel programme *Processing*. Ensuite, nous avons importé notre fichier dans une variable *Processing* (dénommée *police*). A quoi sert une variable (voir section "Les variables") ? Et bien dans ce cas précis, il s'agit d'un nom interne à notre programme qui fait référence au fichier de la police *Georgia-Italic-48.vlw* que nous souhaitons utiliser. A chaque fois que nous écrivons par la suite le mot *police* dans notre programme, *Processing* comprendra qu'il s'agit de faire appel à la police *Georgia Italic 48* contenue désormais dans ce mot. Vous avez peut-être également noté un mot étrange en tout début de cette phrase, le mot *PFont*. Celui-ci indique à *Processing* le genre de la variable et lui permettra d'ouvrir assez de mémoire pour contenir l'ensemble des données de ce genre. Dans le cas de n'importe quelle variable, la syntaxe à utiliser est : {type de la variable} {nom de la variable} = {les valeurs de la variable}. Par exemple, si par un tour de force il était possible d'importer un petit chaton tout mignon dans *Processing* il suffirait d'écrire `petitChaton mioumiou = loadPetitChaton("mioumiou.chat");`. On écrit d'abord le type de la chose, le nom de la chose, et enfin on lui donne sa valeur. Ici, cette valeur est donnée par la fonction `loadFont()` qui va aller chercher la structure géométrique de la police dans le fichier et l'importera dans notre variable nommée "police". La suite de notre programme est probablement un peu plus intuitive. Nous sélectionnons la police avec laquelle nous voulons dessiner. Même s'il n'existe qu'une seule police actuellement dans notre programme, il faut néanmoins passer par cette étape. Notez que vous pouvez importer autant de polices que vous voulez et passer de l'un à l'autre, à l'image de ce qu'il est possible de faire avec les couleurs. Dans cet exemple, nous avons indiqué la taille de la police juste pour

vous montrer qu'il est possible de la changer en cours de route. Enfin, nous dessinons une petite phrase et indiquons la position (x,y) où notre texte doit se dessiner. On obtient le résultat suivant :



Figure 4.32: Une phrase avec la police "Georgia-Italic-48.vlw".

Point d'origine

Pour rendre plus clair le rapport entre la position (x,y) de notre message et sa forme typographique, nous avons également dessiné dans l'illustration ci-dessus une petite croix :

```
size(500,150);
PFont police;
police = loadFont("Georgia-Italic-48.vlw"); textFont(police);
text("Salut tout le monde!", 20, 75); // position du texte
stroke(255,0,0);
line(15,70,25,80);
line(15,80,25,70);
```

Tout comme les rectangles, qui peuvent se dessiner depuis leur point su-

périeur gauche, depuis leur centre, ou depuis ses quatre extrémités, l'écriture du texte peut également être positionnée à partir de plusieurs points d'origine. Par défaut, le texte s'écrit depuis la ligne de base du texte, c'est-à-dire le point en bas à gauche du texte, mais au-dessus des caractères descendants comme les lettres "y" ou "j". Vous pouvez changer la façon dont *Processing* alignera ce texte, en se servant de la fonction `textAlign()` :

```
size(500,250);
PFont police;
police = loadFont("SansSerif-24.vlw");
textFont(police,24);
line(250,0,250,500);
line(0,125,500,125);
textAlign(RIGHT,TOP);
text("right+top", 250, 125);
textAlign(RIGHT,BASELINE);
text("right+baseline", 250, 125);
textAlign(LEFT,CENTER);
text("left+center", 250, 125);
```

4.7 Les transformations

Jusqu'à présent, nous avons dessiné des formes dans la fenêtre de notre application, en nous repérant toujours par rapport au coin supérieur gauche de la fenêtre. Grâce aux transformations, il va être possible de déplacer cette origine, mais aussi de redéfinir l'orientation des axes et même de changer la graduation de ces axes (on parle de changement d'échelle). Par défaut, lorsque l'on dessine une forme (dans notre exemple un rectangle), *Processing* définit le repère suivant :

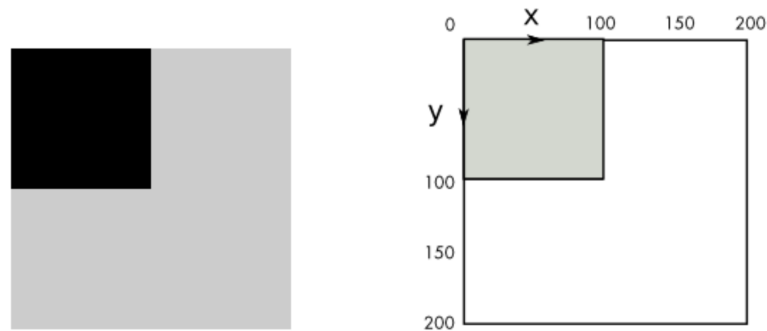


Figure 4.33: Repère défini par *Processing*.

```
size(200, 200);
noStroke();
fill(0);
rect(0, 0, 100, 100);
```

Déplacer

Le changement de la position de l'origine se fait par la commande `translate()`. Nous pouvons nous déplacer sur l'axe x ("horizontalement") et sur l'axe y ("verticalement") et nous allons indiquer à `translate()` de "combien" nous voulons nous déplacer sur chacun des axes. Dans l'exemple suivant, nous déplaçons l'origine de notre repère de 50 pixels en x et de 50 pixels en y. Notons que `translate()` va seulement affecter les formes géométriques qui sont dessinées après cette instruction.

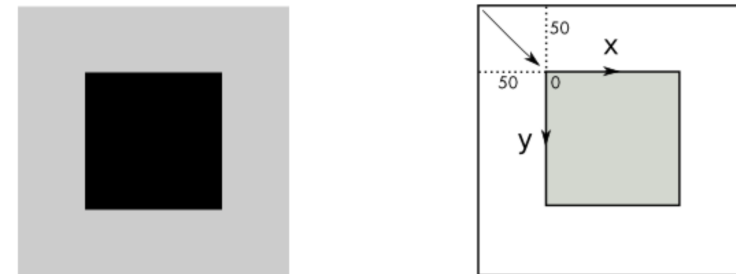


Figure 4.34: L'appel `translate()` affecte les formes géométriques dessinées ensuite.

```
size(200, 200);
noStroke();
fill(0);
translate(50, 50);
rect(0, 0, 100, 100);
```

Enchaîner les `translate()` permet d'accumuler les déplacements comme le montre l'exemple suivant.

```
size(200,200);  
// Noir  
fill(0);  
translate(20,20);  
rect(0,0,40,40);  
  
// Gris  
fill(128);  
translate(60,60);  
rect(0,0,40,40);  
  
// Blanc fill(255);  
translate(60,60);  
rect(0,0,40,40);
```

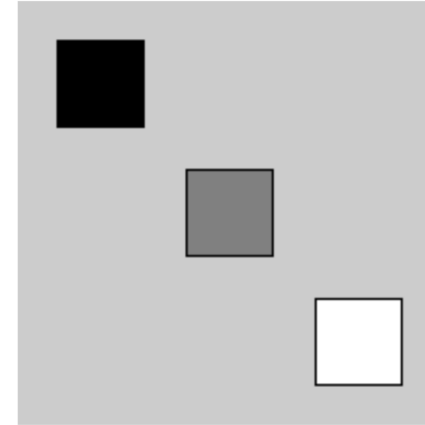


Figure 4.35: *Enchaîner les translate() accumule les déplacements.*

Tourner

Nous avons pu déplacer l'origine du repère de dessin. Nous allons maintenant appliquer une rotation sur les axes de notre repère. Grâce à la commande `rotate()`, les axes x et y peuvent changer d'orientation. La commande `rotate()` prend en paramètre un nombre qui va représenter l'angle de rotation, c'est-à-dire de "combien" nos axes vont tourner par rapport à notre fenêtre. Des valeurs positives indiquent une rotation dans le sens des aiguilles d'une montre.

Deux systèmes de mesure existent pour mesurer un angle : les radians et les degrés. Par défaut, *Processing* travaille en radians mais pour nous il est d'habitude plus facile de raisonner en degrés. Par exemple tourner de 180°, c'est faire un demi-tour. *Processing* permet de passer de transformer une unité en une autre grâce aux fonctions `radians()` et `degrees()`.

```
float d = degrees(PI/4); // transforme des radians en degrés
float r = radians(180.0); // transforme des degrés en radians
```

Illustrons la fonction `rotate()` par un exemple simple. Nous allons faire pivoter un carré autour de l'origine.

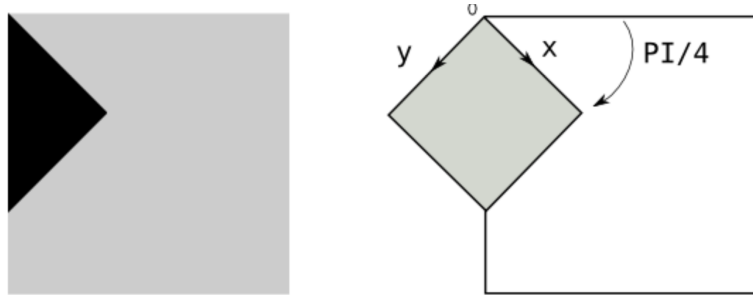


Figure 4.36: *Faire pivoter un carré autour de l'origine.*

```
size(200, 200);
noStroke();
fill(0);
rotate(PI/4);
rect(0, 0, 100, 100);
```

Comme pour `translate()`, `rotate()` se place avant les formes géométriques à dessiner. Il est possible de combiner ces changements d'orientations, qui vont s'accumuler.

```
size(200,200);
smooth();
```

```
translate(width/2, height/2);
for (int i=0;i<360;i+=30) {
  rotate(radians(30));
  quad(0, 0, 30, 15, 70, 60, 20, 60);
  quad(0, 0, 30, 15, 70, 60, 20, 60);
}
```

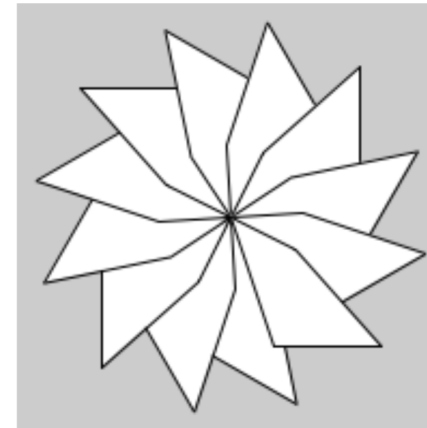


Figure 4.37: *Enchaîner les `rotate()` accumule les changements d'orientations.*

Mettre à l'échelle

La mise à l'échelle permet de redimensionner les objets par la commande `scale()`. Cette fonction permet d'agrandir ou de diminuer la taille des formes géométriques. Elle accepte un ou deux paramètres. Par exemple, `scale(0.5)` va diminuer de moitié la taille des formes géométriques tandis que `scale(2.0)` va la doubler, `scale(1)` n'a aucun effet.

L'écriture avec deux paramètres permet de découpler le redimensionnement en x et en y. Par exemple, `scale(0.5, 2.0)` va écraser la forme sur les x de moitié tandis que sur les y sa taille sera doublée.

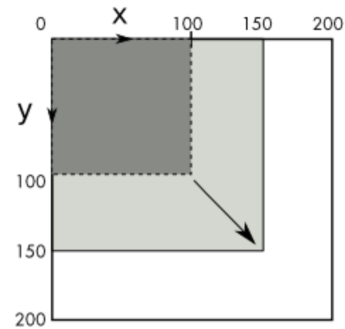


Figure 4.38: *Mise à l'échelle égale en x et y.*

```
size(200,200);  
scale(1.5);  
rect(0,0,100,100);
```

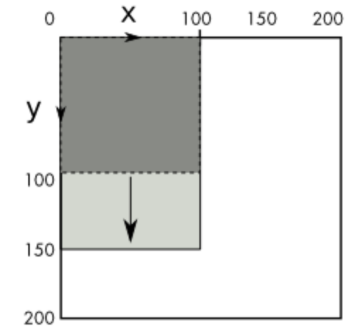


Figure 4.39: *Mise à l'échelle différente en x et y.*

```
size(200,200);  
scale(1.0,1.5);  
rect(0,0,100,100);
```

Comme pour `rotate()` et `translate()`, l'enchaînement de `scale()` permet d'accumuler les mises à l'échelle. Illustrons cette propriété par le sketch suivant, qui reprend l'idée des poupées russes en emboitant des carrés par le jeu des `scale()` successifs.

Figure 4.40: *Enchaîner les scale() accumule les mises à l'échelle.*

```
size(200,200);
noStroke();
// Noir
fill(0);
scale(1);
rect(0,0,200,200);

// Gris
fill(128);
scale(0.5);
rect(0,0,200,200);

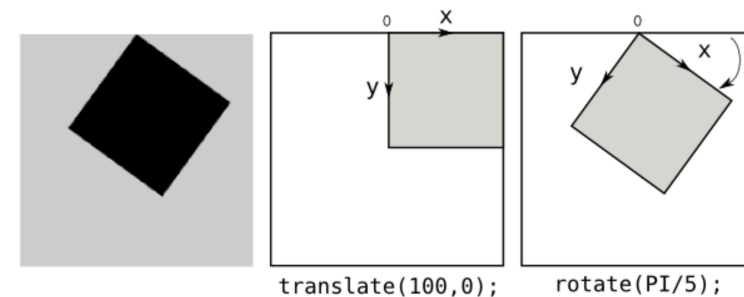
// Blanc
fill(255);
```

```
scale(0.5);
rect(0,0,200,200);
```

L'ordre des transformations

Il est possible de combiner plusieurs types de transformations. Comme nous l'avons vu dans les exemples précédents, les transformations s'accumulent au fur et à mesure des appels successifs à `translate()`, `rotate()` ou `scale()` et chacune des transformations tient compte des transformations précédentes.

Lorsqu'on utilise plusieurs types de transformations, leur ordre d'écriture va être important. Lorsque vous êtes en voiture, "tourner à gauche" puis "continuer tout droit" est différent de "continuer tout droit" puis "tourner à gauche". Vous n'arrivez pas forcément au même endroit en suivant successivement ces deux instructions. C'est la même chose pour les transformations dans *Processing*. Illustrons ceci par un exemple en inversant un `translate()` et un `rotate()`.

Figure 4.41: *D'abord translate() puis rotate().*

```
size(200,200);
```

```
smooth();
fill(0);
translate(100,0);
rotate(PI/5);
rect(0,0,100,100);
```

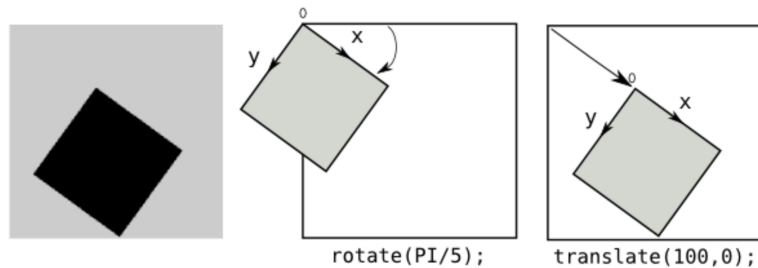


Figure 4.42: D'abord `rotate()` puis `translate()`.

```
size(200,200);
smooth();
fill(0);
rotate(PI/5);
translate(100,0);
rect(0,0,100,100);
```

Isoler les transformations

Nous venons de voir que les transformations s'accumulaient au fur et à mesure de l'utilisation des commandes `translate()`, `rotate()` et `scale()`. Nous allons voir à présent comment sauvegarder les transformations à un moment donné et comment les restaurer ensuite, au cours d'une animation interactive

faisant appel à la méthode `draw()`.

Nous allons utiliser pour cela deux fonctions qui s'utilisent toujours par paire : `pushMatrix()` et `popMatrix()`. Nous verrons en fin de chapitre pourquoi ces deux fonctions portent un nom si bizarre. Pour les deux exemples qui suivent, nous allons identifier les transformations suivantes :

- A : origine en haut à gauche de la fenêtre.
- B : origine au centre de l'écran.
- C : origine au centre de l'écran, rotation de $\pi/4$.

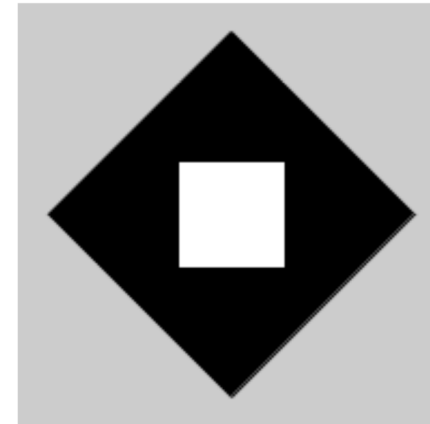


Figure 4.43: Sauvegarde de A, Transformation B, Restauration de A.

```
size(200,200);
smooth();
rectMode(CENTER);
```

```
// Repère au centre de l'écran
translate(width/2,height/2);
```

```
// Sauvegarde de A
pushMatrix();

// Transformation B
rotate(PI/4);

// Dessin du carré noir
fill(0);
rect(0,0,120,120);

// Restauration A
// A ce point-là, notre repère revient au centre de l'écran
popMatrix();

// Dessin du carré blanc
// qui ne tient pas compte de la rotation
fill(255);
rect(0,0,50,50);
```

Il est possible d'imbriquer les sauvegardes de transformations, c'est-à-dire qu'à l'intérieur de n'importe quelle paire de `pushMatrix()/popMatrix()` nous pouvons rappeler ces fonctions pour sauver l'état de la transformation courante. Reprenons l'exemple précédent en plaçant une paire `pushMatrix()/popMatrix()` qui encadre la première transformation `translate(width/2, height/2)`.

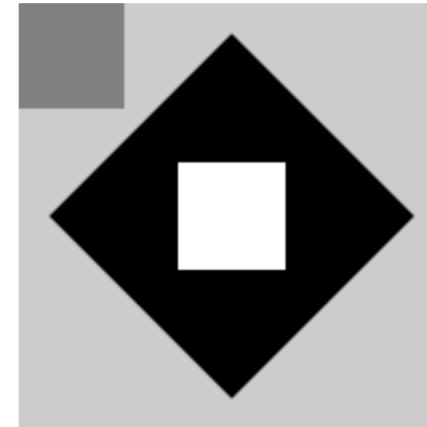


Figure 4.44: *Sauvegarde de A, Transformation B, Sauvegarde de B, Transformation C, Restauration de B, Restauration de A.*

```
size(200,200);
smooth();
rectMode(CENTER);
noStroke();

// Sauvegarde de A
pushMatrix();

// Transformation B
translate(width/2,height/2);

// Sauvegarde de B
pushMatrix();
```

```
// Transformation C
rotate(PI/4);

// Dessin du carré noir
fill(0);
rect(0,0,120,120);

// Restauration de B
popMatrix();

// Dessin du carré blanc
// qui ne tient pas compte de la rotation rotate(PI/4)
fill(255);
rect(0,0,50,50);

// Restauration de A
popMatrix();

// Dessin du carré gris
fill(128);
rect(0,0,100,100);
```

Transformer en 3D

Toutes les transformations que nous venons d'aborder sont applicables en trois dimensions (3D). *Processing* permet de passer en 3D au moment de l'appel à `size()` :

```
size(300,300,P3D);
```

Dans ce mode, *Processing* définit un axe z qui pointe vers le fond de l'écran.

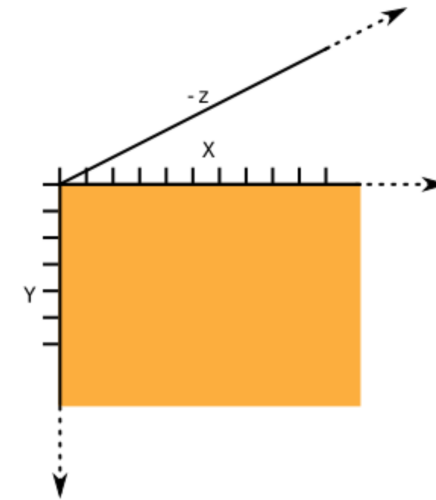


Figure 4.45: Un axe z qui pointe vers le fond de l'écran.

Les transformations de déplacement et de mise à l'échelle vont s'écrire en intégrant un troisième paramètre. Par exemple, pour se déplacer au centre de l'écran le long de x et y puis dessiner les formes comme si elles étaient éloignées de nous, nous pourrions écrire la ligne de code suivante : `translate(width/2, height/2, -100)` ; Pour les rotations, nous disposons de trois fonctions : `rotateX`, `rotateY` et `rotateZ` qui permettent respectivement de tourner autour des axes x, y et z. *Processing* intègre des fonctions de dessin de formes simples en 3D, notamment les cubes et les sphères. Nous allons créer un cube dont la rotation autour des axes x et y va être paramétré par la position de la souris.

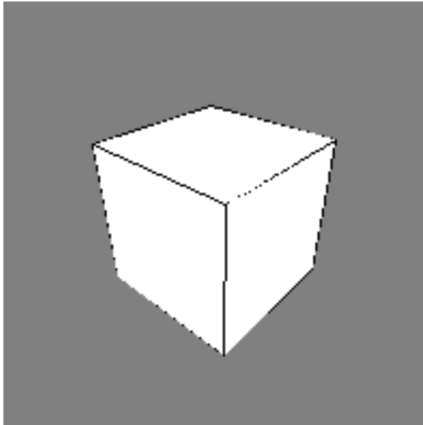


Figure 4.46: Un cube dont la rotation autour des axes x et y va être paramétré par la position de la souris.

```
float rx = 0;
float ry = 0;
float z = 100;

void setup() {
  size(200,200,P3D);
}

void draw() {
  background(128);
  rx = map(mouseX, 0,width,-PI,PI);
```

```
  ry = map(mouseY, 0,height,-PI,PI);
  translate(width/2,height/2,z);
  rotateX(rx);
  rotateY(ry);
  box(30);
}
```

Dans ce sketch, nous avons introduit une nouvelle fonction `map()`, qui permet de transformer une valeur d'une plage de valeurs à une autre plage de valeurs. Voici un exemple simple pour expliquer ce concept :

```
float v = 100;
float m = map(v,0,200, 0,1); // m vaut 0.5
```

Dans cet exemple, `map()` va transformer la valeur 100 qui est dans l'intervalle `[0;200]` et calculer la valeur équivalente dans l'intervalle `[0;1]`. La valeur 0.5 est retournée dans `m`. Dans notre sketch, cette fonction permet de transformer la valeur de `mouseX` dans l'intervalle situé entre 0 et `width` en une valeur équivalente dans l'intervalle entre `-PI` et `PI`.

Les concepts de `pushMatrix()` et `popMatrix()` sont aussi applicables en 3D pour sauvegarder et restaurer les transformations. C'est le meilleur moyen pour dessiner des univers en 3D, contenant plusieurs objets en mouvement les uns par rapport aux autres sans avoir recours à des concepts mathématiques complexes.

Toutes les transformations dans *Processing* sont stockées dans un tableau de 16 nombres qui est appelé matrice ou *matrix* en anglais. Ces nombres sont directement modifiés par des appels aux fonctions de transformations. Si vous êtes curieux, vous pouvez imprimer ce tableau par la fonction `printMatrix()`.

Programmer avec *Processing*

Sommaire

5.1 Les variables	47
5.2 Les conditions	51
5.3 Les répétitions	53
5.4 Les listes	57
5.5 Les méthodes	63
5.6 Les objets	69
5.7 Les commentaires	73

Processing est basé sur le langage Java. C'est cette syntaxe qui sera utilisée lorsque vous allez programmer. *Processing* vous facilite la maîtrise de ce langage en se chargeant de manière transparente des opérations relativement complexes comme gérer les fenêtres, le son, la vidéo, la 3D et bien d'autres choses encore. Ce logiciel propose une large palette de fonctionnalités prédéfinies qui simplifie la conception de programmes créatifs, notamment pour les personnes ne maîtrisant pas les notions complexes de programmation et de mathématiques.

5.1 Les variables

42	491	33.145	5	« Salut tout le monde ! »	true
x	y	angle	i	message	vie

Figure 5.1: Dans la mémoire de l'ordinateur, chaque variable a un compartiment dont la taille est fonction du type d'information.

Une variable est une donnée que l'ordinateur va stocker dans l'espace de sa mémoire. C'est comme un compartiment dont la taille n'est adéquate que

pour un seul type d'information. Elle est caractérisée par un nom qui nous permettra d'y accéder facilement.

Il existe différents type de variables : des nombres entiers (`int`), des nombres à virgule (`float`), du texte (`String`), des valeurs vrai/faux (`boolean`). Un nombre à décimales, comme 3,14159, n'étant pas un nombre entier, serait donc du type `float`. Notez que l'on utilise un point et non une virgule pour les nombres à décimales. On écrit donc 3.13159. Dans ce cas, les variables peuvent être annoncées de cette manière :

```
float x = 3.14159;  
int y = 3;
```

Le nom d'une variable peut contenir des lettres, des chiffres et certains caractères comme la barre de soulignement. à chaque fois que le programme rencontre le nom de cette variable, il peut lire ou écrire dans ce compartiment. Les variables qui vont suivre vous donneront des exemples simples de leur utilisation. Pour résumer, une variable aura un type, un nom et une valeur qui peut être lue ou modifiée.

int

Dans la syntaxe de *Processing*, on peut stocker un nombre entier, par exemple 3, dans une variable de type `int`.

```
int entier;  
entier = 3;  
print(entier);
```

float

La variable `float` est un nombre avec décimales, par exemple 2,3456.

```
float decimal;  
decimal = PI;  
print(decimal);
```

double

La variable `double` est également de nombre avec décimales, mais qui fournissent davantage de précision que le type `float`.

```
double long_decimal;  
long_decimal = PI;  
print(long_decimal);
```

boolean

La variable `boolean` est un type de variable qui ne connaît que deux états : Vrai (`true`) ou Faux (`false`). Elle est utilisée dans les conditions pour déterminer si une expression est vraie ou fausse.

```
boolean vraifaux;  
vraifaux = true;  
println(vraifaux);
```

char

La variable `char` sert à stocker un caractère typographique (une lettre). Notez l'usage de ce qu'on appelle des guillemets simples.

```
char lettre;  
lettre = 'A';  
print(lettre);
```

string

La variable `string` sert à stocker du texte. Notez l'usage des guillemets doubles.

```
String texte;  
texte = "Bonjour!";  
print(texte);
```

color

La variable `color` sert à stocker une couleur. Cette variable est utile lorsqu'on veut réutiliser souvent les mêmes couleurs.

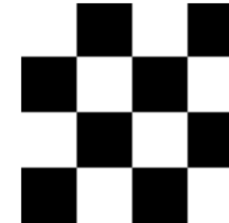


Figure 5.2: La variable `color` permet de stocker une couleur qu'on veut réutiliser.

```
noStroke();  
color blanc = color(255, 255, 255);  
color noir = color(0, 0, 0);  
  
fill(blanc); rect(0, 0, 25, 25);  
fill(noir); rect(25, 0, 25, 25);  
fill(blanc); rect(50, 0, 25, 25);  
fill(noir); rect(75, 0, 25, 25);  
  
fill(noir); rect(0, 25, 25, 25);  
fill(blanc); rect(25, 25, 25, 25);  
fill(noir); rect(50, 25, 25, 25);  
fill(blanc); rect(75, 25, 25, 25);  
  
fill(blanc); rect(0, 50, 25, 25);  
fill(noir); rect(25, 50, 25, 25);  
fill(blanc); rect(50, 50, 25, 25);  
fill(noir); rect(75, 50, 25, 25);
```

```
fill(noir); rect(0, 75, 25, 25);  
fill(blanc); rect(25, 75, 25, 25);  
fill(noir); rect(50, 75, 25, 25);  
fill(blanc); rect(75, 75, 25, 25);
```

5.2 Les conditions

Les conditions donnent une part d'autonomie à votre ordinateur. Elles lui permettent de modifier le comportement du programme en fonction de diverses conditions de votre choix. Par exemple, si vous vouliez changer l'apparence de votre programme en fonction de l'heure, vous pourriez lui demander d'avoir un fond noir entre 10 heures du soir et 6 heures du matin et un fond blanc le reste du temps. C'est ce questionnement - "Quelle heure est-il?" - qui constitue la condition. "S'il fait nuit, je dois dessiner un fond noir, sinon je dessine un fond blanc" pourrait constituer en quelque sorte le dialogue interne de *Processing* lorsqu'il rencontre une condition.

Comparaison

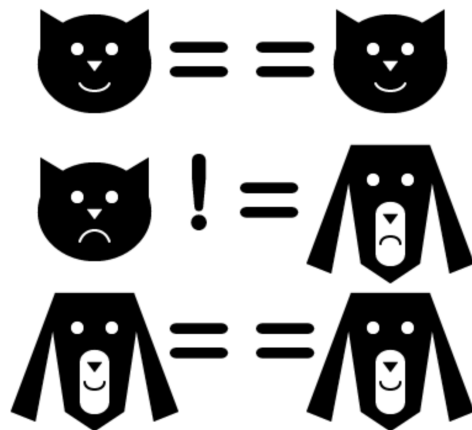


Figure 5.3: La condition se base sur la comparaison.

La base de la condition, c'est la comparaison. Avant de pouvoir agir selon

une condition, il faut d'abord formuler la question que *Processing* doit se poser. Cette question sera quasiment toujours une question de comparaison. Si le résultat de la question est "oui", *Processing* exécutera une suite d'instructions. Si la réponse est "non", il exécutera une autre. Dans *Processing*, ce "oui"/"non" s'écrit "true" et "false".

La syntaxe d'une condition est la suivante : `if (TEST)` . Le TEST correspond à l'opération (égalité, plus petit, plus grand) que vous aurez choisie pour comparer deux valeurs et déterminer si la réponse à la question est true ou false. Si la réponse est true, *Processing* exécutera les instructions entre les deux accolades. L'instruction else permet de gérer le cas de figure dans lequel la condition n'est pas validée. Elle exécute elle aussi tout ce qui se trouve à l'intérieur de ses accolades. Vous pouvez mettre autant d'instructions que vous voulez entre ces deux types d'accolades.

Egalité Pour vérifier l'égalité de deux valeurs, on utilise la formule suivante : `if (valeur1 == valeur2) { }`. L'exemple suivant écrit "Il est midi" dans la console si la méthode `hour()` donne la valeur 12.

```
if (hour() == 12) {  
    println("Il est midi!");  
} else {  
    println("Ce n'est pas midi!");  
}
```

Résultat de l'application exécutée entre 12h00 et 12h59 est : Il est midi !

Plus petit que et plus grand que On peut vérifier qu'une valeur est plus petite ou plus grande qu'une autre en utilisant les opérateurs `<` et `>`. L'exemple suivant va écrire dans la console si nous sommes le matin ou non.

```
if (hour() < 12) {  
  println("C'est le matin!");  
} else {  
  println("Ce n'est pas le matin!");  
}
```

Résultat de l'application exécutée après 12h59 : Ce n'est pas le matin

Combiner les décisions

Les if et else peuvent être combinés pour gérer plusieurs cas de figure.

```
if (hour() < 12) {  
  println("C'est le matin!");  
} else if (hour() == 12) {  
  println("Il est midi!");  
} else {  
  println("Ce n'est pas le matin!");  
}
```

Résultat de l'application exécutée avant 12h00 : Ce n'est pas le matin !

Combiner les tests

Plusieurs tests peuvent être combinés au sein d'une même décision pour rendre le choix plus précis. Les opérateurs && (et) ainsi que || (ou) permettent de combiner des tests. Par exemple pour déterminer si nous sommes la nuit ou le jour, nous avons besoin de trier les heures qui sont tard le soir et tôt le matin de celles du reste de la journée :

```
if (hour() < 6 && hour() > 20) {  
  println("Il fait nuit!");  
} else {  
  println("Il ne fait pas nuit!");  
}
```

Résultat de l'application exécutée à 16h50 : Il ne fait pas nuit !

5.3 Les répétitions

Les répétitions permettent d'exécuter une série d'instructions plusieurs fois de suite. Elles évitent de dupliquer inutilement des portions de code. Attention ! les répétitions ne permettent pas de créer des animations dans le temps (d'autres instructions existent pour cela) ! Lorsque l'ordinateur lit le programme et rencontre une boucle, il va exécuter instantanément autant de fois de suite le code écrit dans le bloc de la boucle qu'on lui a indiqué.

L'exemple ci-dessous va nous permettre d'illustrer simplement cette notion. L'objectif est de réaliser un dessin affichant dix lignes noires horizontales. Le premier code contient dix fois l'instruction `line()`, le second code est réalisé à partir d'une boucle. Le résultat des deux codes est le même, la différence se situant au niveau de la longueur du code, l'un étant plus rapide à saisir (et ultérieurement à modifier) que l'autre.

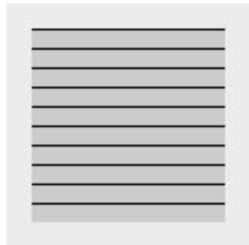


Figure 5.4: Un dessin affichant 10 lignes noires horizontales

```
line(0, 0, 100, 0);  
line(0, 10, 100, 10);  
line(0, 20, 100, 20); line(0, 30, 100, 30);  
line(0, 40, 100, 40);  
line(0, 50, 100, 50);
```

```
line(0, 60, 100, 60);  
line(0, 70, 100, 70);  
line(0, 80, 100, 80);  
line(0, 90, 100, 90);
```

...ou plus simplement :

```
for (int i = 0; i < 100; i = i + 10) {  
    line(0, i, 100, i);  
}
```

La boucle for

Ce type de boucle permet de répéter une série d'instructions un nombre de fois défini. Elle incorpore une variable qui va s'incrémenter à chaque passage dans la boucle. On utilise souvent `i` comme nom pour la variable interne de la boucle. Elle comprend : un nombre de départ, un nombre maximal et une incrémentation. Sa syntaxe est la suivante : `for (int i = NombreDeDépart; i < NombreMaximal; i = i + INCREMENT) { }`.

L'exemple de code ci-dessous va afficher des rectangles blancs côte à côte qui auront chacun 10 pixels de large sur toute la largeur de l'espace de dessin. Ils seront espacés de 5 pixels. Nous allons afficher le premier rectangle aux coordonnées 0,0. Les suivants seront affichés aux coordonnées 15,0 puis 30,0 et ainsi de suite. Notre boucle va incrémenter sa variable de 15 pixels à chaque étape. Comme nous voulons remplir toute la largeur de l'espace de dessin, la valeur maximale sera égale à la largeur (width) de cette fenêtre de visualisation.

```
for (int i = 0; i < width; i = i + 15) {  
  rect(i, 0, 10, 10);  
}
```



Figure 5.5: Un dessin affichant des rectangles de taille 10 pixels sur la largeur de la fenêtre de visualisation.

Les compteurs

Jusqu'à maintenant, nous avons utilisé les boucles for de manière à ce que la variable interne de la boucle soit directement exploitable. Dans l'exemple précédent, elle nous donne immédiatement la valeur exacte de la position sur l'axe x du rectangle à dessiner.

Les boucles for peuvent aussi être utilisées comme des compteurs. On leur donne un minimum, un maximum et on incrémente la variable seulement de 1 à chaque étape de la boucle : ceci permet de compter le nombre de fois que les instructions seront exécutées, d'où l'expression de compteur. Cette méthode implique davantage de calculs mathématiques pour exploiter la variable au sein de la boucle.

```
for (int i = 0; i < 5; i = i + 1) {  
  rect(i, 0, 10, 10);  
}
```



Figure 5.6: Un dessin affichant des rectangles de taille 10 pixels : les rectangles sont empilés les uns sur les autres.

Nous voyons dans l'exemple ci-dessus que les rectangles sont empilés les uns sur les autres. Puisque notre variable *i* aura des valeurs de 0 à 5 et que nous l'utilisons telle quelle pour placer les rectangles dans l'espace, ceux-ci seront placés aux coordonnées 0,0 ; 1,0 ; 2,0 ; 3,0 ; 4,0. Pour obtenir le même résultat que l'exemple précédent, il faut multiplier la variable. Dans le cas présent nous allons la multiplier par 15 (10 pour la largeur du rectangle et 5 de marge). Ils sont ainsi disposés tous les 15 pixels. Nous pourrions également utiliser le chiffre du compteur pour réaliser d'autres opérations.



Figure 5.7: Un dessin affichant des rectangles de taille 10 pixels : les rectangles sont correctement placés.

```
for (int i = 0; i < 5; i = i + 1) {
    rect(i * 15, 0, 10, 10);
}
```

Imbriquer des boucles

Les boucles peuvent s’imbriquer les une dans les autres. Cette technique permet de rapidement passer à des visualisations à deux, voir trois dimensions. Lorsqu’on imbrique des boucles, il faut prendre garde au nom que l’on donne à la variable de chaque boucle. En effet si elles se nomment toutes *i*, le programme va mélanger les boucles. Chaque variable de chaque boucle doit avoir un nom propre. Par exemple : *i*, *j*, *k*, etc. ou si elles sont liées à des dimensions : *x*, *y* et *z*.

Boucle de départ On la définit ainsi :

```
translate(7, 7);
for (int x = 0; x < 6; x = x + 1) {
```

```
    rect(x * 15, 0, 10, 10);
}
```



Figure 5.8: Une seule boucle for : des rectangles de taille 10 pixels sur la largeur de la fenêtre de visualisation.

Deux boucles On construit 2 boucles :

```
translate(7, 7);

//Première boucle (hauteur)
for (int y = 0; y < 6; y = y + 1) {
    //Seconde boucle (largeur)
    for (int x = 0; x < 6; x = x + 1) {
        rect(x * 15, y * 15, 10, 10);
    }
}
```

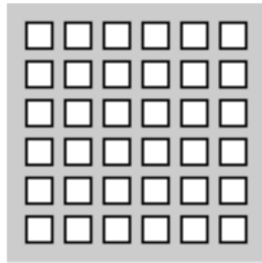


Figure 5.9: Deux boucle for : des rectangles de taille 10 pixels sur la largeur et la hauteur de la fenêtre de visualisation.

Trois boucles Dans cet exemple nous introduisons un espace 3D. Pour placer nos rectangles dans la profondeur, nous devons utiliser la méthode `translate()`.

```
size(100, 100, P3D);
translate(7, 7);

//Première boucle (profondeur)
for (int z = 0; z < 6; z = z + 1) {
  translate(0, 0, z * -15); //On recule l'objet sur l'axe z
  //Seconde boucle (hauteur)
  for (int y = 0; y < 6; y = y + 1) {
    //Troisième boucle (largeur)
    for (int x = 0; x < 6; x = x + 1) {
      rect(x * 15, y * 15, 10, 10);
    }
  }
}
```

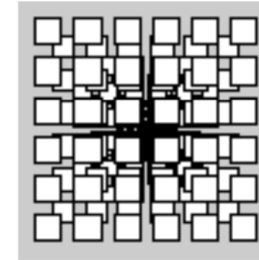


Figure 5.10: Trois boucles for : des rectangles de taille 10 pixels sur la largeur, la hauteur et la profondeur de la fenêtre de visualisation.

Variations Voici une série de variations des exemples ci-dessous qui utilisent les méthodes `fill()`, `scale()` ou `rotate()`. A vous d'expérimenter les transformations au sein de boucles.

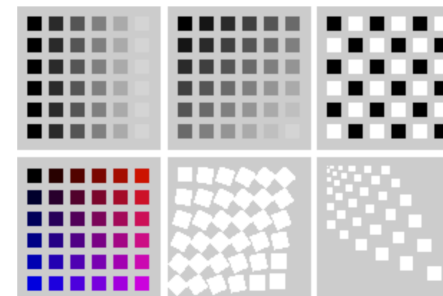


Figure 5.11: Series de variations utilisant la boucle for.

5.4 Les listes

On peut mettre de nombreux genres de choses dans une variable : un chiffre, un chiffre à virgule, la phrase d'un texte, voire même toute une image ou tout un morceau de son. Mais bien que les variables puissent théoriquement contenir presque tout type de valeur, elles ne peuvent contenir qu'une seule de ces valeurs à la fois. Dans certains cas, il serait pratique d'avoir plusieurs choses regroupées, au moins du même genre, dans une seule entité. C'est pour cette raison qu'un genre très particulier de variables a été inventé, *les listes*.

Les listes permettent de stocker un nombre fixé d'avance de données ou d'objets dans une même variable. Au lieu de créer 20 variables pour stocker 20 valeurs différentes d'un même genre, nous pouvons créer un seul contenant pour ces 20 valeurs et y accéder une par une via cette seule et unique variable.

Créer une liste

Si nous utilisons des éléments dits fondamentaux, comme les chiffres, il est très facile de fabriquer une liste :

```
int[] numbers = 90,150,30;
```

Le signe du double crochet signifie qu'il ne s'agit plus d'une variable avec un seul entier à l'intérieur, mais d'une liste d'entiers avec plusieurs valeurs à l'intérieur. Ensuite nous remplissons cette liste dénommée *numbers* (nous aurions très bien pu lui donner un autre nom) avec les valeurs notées entre les accolades.

L'ordinateur créera assez d'emplacements dans la mémoire et placera chacune des valeurs dans les cases correspondantes. C'est d'ailleurs pour cette

raison qu'il faut indiquer le mot *int*, car *Processing* a besoin de connaître la taille de chaque case de la liste (dans ce cas précis, nous lui indiquons à l'aide de cette instruction qu'il s'agit de nombres entiers). S'il s'agissait d'images, comme on le verra plus loin, chaque case de la liste aurait besoin de davantage de place.

Notez que sur certains claviers francophones ou sur certaines plateformes, il est parfois difficile de localiser le crochet d'ouverture "[" et de fermeture "]". Sur un clavier français d'ordinateur Apple, par exemple, il faut appuyer en même temps sur les 3 touches `{alt} + {maj} + (` pour le crochet d'ouverture, et `{alt} + {maj} + )` pour le crochet de fermeture. Par ailleurs, on doit taper `{alt} + (` pour l'accolade d'ouverture et `{alt} + )` pour l'accolade de fermeture.

Créer une liste vide

Bien qu'elle soit lisible, cette méthode de création directe ne marchera pas avec des éléments comme les sons ou les images. A cause de cette limitation, vous risquez d'utiliser rarement cette méthode. Déclarer une liste se fera donc le plus souvent de la manière suivante : `type[] nomDeLaListe = new type[NOMBRE D'ÉLÉMENTS]`. L'exemple ci-dessous crée une liste de trois nombres entiers. Attention ! Au moment où la liste dénommée *numbers* est créée, celle-ci est constituée d'une suite de cases vides qui ne contiennent aucune valeur.

```
int[] numbers = new int[3];
```

Le chiffre 3 entre accolades indique que nous créons une liste de 3 cases actuellement vides.

Remplir la liste

Placer des valeurs dans une liste fonctionne de la même manière qu'assigner une valeur à une variable. Il faut en plus préciser à quelle position de la liste on ajoute la valeur.

```
numbers[0] = 90 ;  
numbers[1] = 150 ;  
numbers[2] = 30 ;
```

La première position de la liste commençant par 0, si nous créons une liste de 3 valeurs, les cases les contenant seront donc numérotées de 0 à 2. On passe d'une position de la liste à une autre en l'incrémentant de 1.

Utiliser le contenu d'une liste

Utiliser l'élément d'une liste est similaire également à l'utilisation d'une variable. Il faut juste préciser la position de l'élément en question :

```
println( numbers[0] );
```

A nouveau, notez que l'ordinateur commence à compter à la position 0 et non pas 1. Si nous demandons la valeur `numbers[2]`, *Processing* nous donnera la valeur à la troisième position et non pas à la deuxième. Un bon moyen pour se rappeler cette particularité, c'est de considérer que *Processing* compte de la manière suivante : *zéroième, premier, deuxième, troisième*, etc. La *zéroième* valeur concerne la valeur au début de la liste.

Dans l'exemple ci-dessous nous utilisons une boucle pour calculer la somme de tous les éléments de la liste déclarée précédemment :

```
int[] numbers = new int[3];
```

```
numbers[0] = 90 ;  
numbers[1] = 150 ;  
numbers[2] = 30 ;
```

```
int somme = 0;
```

```
for (int i = 0; i < 3; i++) {  
    somme = somme + numbers[i];  
}  
println(somme);
```

Le programme devrait afficher le résultat dans votre console, tout en bas de la fenêtre *Processing*. Quelques éléments supplémentaires d'explication pour mieux comprendre le fonctionnement de ce programme. Tout d'abord, nous créons une variable qui contiendra la somme de toutes les valeurs. Elle débute à zéro. Ensuite, nous enclenchons une boucle qui se répétera 3 fois, en additionnant à chaque fois la prochaine valeur à la somme. Ici nous pouvons voir la relation étroite qui existe entre une liste et une boucle `for()`. La valeur `i` permet de passer une à une dans chacune des valeurs de la liste, en commençant avec la position zéro, ensuite la position 1, et enfin la position 2.

Si vous vous mettez à analyser le code de programmes consultables en ligne, par exemple dans les sketches librement mis à disposition sur *OpenProcessing*, vous verrez beaucoup de code avec `"quelqueChose[i]"`.

Une suite d'images

Un des usages les plus pratiques des *listes* concerne l'importation de médias dans *Processing*. Imaginons que nous voulons importer cinq images dans *Processing* destinées ensuite à être utilisées dans un sketch. Sans employer les listes, il faudrait écrire quelque chose comme ceci :

```
PImage photo1;
photo1 = loadImage("photo_1.png");
image(photo1,0,0);

PImage photo2;
photo2 = loadImage("photo_2.png");
image(photo2,50,0);

PImage photo3;
photo3 = loadImage("photo_3.png");
image(photo3,100,0);

PImage photo4;
photo4 = loadImage("photo_4.png");
image(photo4,150,0);

PImage photo5;
photo5 = loadImage("photo_5.png");
image(photo5,200,0);
```

Certes, cet exemple reste encore relativement gérable dans la mesure où vous faites appel à seulement 5 images. Cela étant, les occasions d'erreurs de sai-

sie sont nombreuses. En écrivant ces quelques lignes pour ce manuel, nous-mêmes avons plusieurs fois effectué des erreurs de frappe, notamment en oubliant de changer un chiffre que nous venions de copier d'une ligne précédente !

Une meilleure façon d'écrire cet exemple serait d'utiliser des listes :

```
PImage[] images = new PImage[5];

images[0] = loadImage("image_0.png");
images[1] = loadImage("image_1.png");
images[2] = loadImage("image_2.png");
images[3] = loadImage("image_3.png");
images[4] = loadImage("image_4.png");

image( images[0], 0, 0);
image( images[1], 50, 0);
image( images[2], 100, 0);
image( images[3], 150, 0);
image( images[4], 200, 0);
```

En utilisant une liste, on peut mettre toutes nos images dans une seule variable qui doit être initialisée qu'une seule fois : `PImage[] images = new PImage[5]`. Ensuite, il suffit de remplir chaque valeur de fichier pour chacun des emplacements dans la liste. Mais même cette écriture est trop longue. Comment allez-vous faire, par exemple, lorsque vous aurez 200 images, voire 2.000 ? Allez-vous vraiment écrire 200 fois toutes ces lignes ? Et comment ferez-vous lorsque vous voudrez changer le nom des fichiers importés ?

Voici la meilleure manière d'importer une suite d'images :

```
size(500,500);
PImage[] images = new PImage[20];
for(int i=0; i<images.size(); i++) {
    images[i] = loadImage("image_" + i + ".png");
    image( images[i], random(width), random(height) );
}
```

En utilisant une répétition `for()`, nous pouvons désormais importer autant d'images que nous voulons. Tout d'abord, nous créons la liste d'images. Il s'agit au départ d'une liste vide :

```
PImage[] images = new PImage[20];
```

Ensuite nous récupérons la longueur de cette liste en demandant à la liste elle-même combien d'éléments elle contient via `images.size()` et utilisons cette longueur dans la répétition `for()`.

```
for(int i=0; i<images.size(); i++)
```

Nous pouvons même automatiser l'importation des images en utilisant la valeur `i` pour composer le nom du fichier des images dans le dossier data. En utilisant le signe `"+"`, on peut concaténer deux mots ensemble, ce qui veut dire que nous allons assembler deux mots/signes en un seul message.

```
images[i] = loadImage("image_" + i + ".png");
```

En concaténant les mots `"image_"`, la valeur de la variable `i`, et `".png"`, nous ob-

tenons un message qui ressemblerait à quelque chose comme `"image_42.png"`. De cette manière, nous pouvons utiliser une seule ligne de code pour faire rentrer autant d'images que nous voulons. Si la variable `i` contient la valeur de 9, le nom du fichier importé sera `image_9.png`. Si la variable `i` contient la valeur 101, le nom du fichier importé sera `image_101.png`. Pour que cette technique marche, il faut juste préparer au préalable des fichiers d'images dans votre dossier data. Une fois chacune de ces images importées dans notre programme, nous dessinons l'image quelque part dans le sketch en écrivant :

```
image( images[i], random(width), random(height) );
```

A l'aide de l'instruction `random`, cette ligne de code affiche une image (0, 1, 2, 3, ...) qui sera placée à chaque fois de façon aléatoire sur l'axe x et sur l'axe y de l'espace de dessin. Les deux valeurs `width` et `height` permettent de connaître automatiquement la taille de la fenêtre de visualisation utilisée par le sketch. En faisant appel à elles, on précise les limites verticales et horizontales maximales où doivent se positionner les images, l'instruction `random()` ayant pour fonction de générer un nombre au hasard ne dépassant pas la valeur mentionnée entre ses parenthèses.

Une suite de pixels

Il se trouve que vous connaissiez déjà les listes et que vous vous en serviez déjà sans le savoir. Par exemple, si vous avez lu le chapitre sur les images vous savez déjà qu'une variable de type `PImage` ne contient pas une seule valeur mais plusieurs, à moins que votre image ne fasse qu'un pixel de large et un pixel de haut. A part cette exception rare, votre image contient un ensemble de valeurs qui représentent l'ensemble des pixels qui doivent former l'image. C'est cet ensemble qui s'appelle une liste.

Vous pouvez accéder directement aux pixels d'une image en demandant un accès direct à sa sous-variable nommée "pixels". Imaginons que nous avons au départ l'image d'une mystérieuse dame dans un fichier nommé *lhooq.png*.



Figure 5.12: L'image *lhooq.png*.

Si nous importons cette image dans *Processing*, nous pouvons l'utiliser pour récupérer la couleur d'un pixel en particulier en entrant à l'intérieur de sa liste de couleurs. Cette liste s'appelle `pixels[]`.

```
size(512,256);
```

```
PImage lhooq;  
lhooq = loadImage("lhooq.png");  
image(lhooq,0,0);
```

```
int x = 119;
```

```
int y = 119;  
  
int index = x + (y * lhooq.width);  
color c = lhooq.pixels[index];  
  
noStroke();  
fill(c);  
rect(256,0,256,256);  
  
stroke(0);  
line(x-5,y,x+5,y);  
line(x,y-5,x,y+5);
```

Les deux lignes les plus importantes sont :

```
int index = x + (y * lhooq.width);  
color c = lhooq.pixels[index];
```

Nous vous rappelons qu'une image n'est rien d'autre qu'une liste de pixels, une image étant notamment composée non pas d'une mais de plusieurs couleurs. Dans *Processing*, les variables de type `PImage` servent d'ailleurs à stocker les valeurs de ces pixels dans une longue liste linéaire. Ce que nous voyons comme une liste à deux $\{x,y\}$ dimensions, `PImage` la voit en une seule dimension. Si nous voulons trouver une valeur à la position $\{2,1\}$, en réalité il faut indiquer que nous voulons la position 17 de l'image. En effet, entre un point sur une ligne et le point correspondant sur la prochaine ligne, il existe 16 pixels. Autrement dit, `PImage` doit compter 16 pixels chaque fois qu'il veut descendre une ligne. C'est pour cette raison que la formule pour identifier un pixel dans

une image s'écrit $\{x + (\text{largeur image} * y)\}$. Pour chaque ligne y , il existe une largeur de valeurs x qu'il faut dépasser.

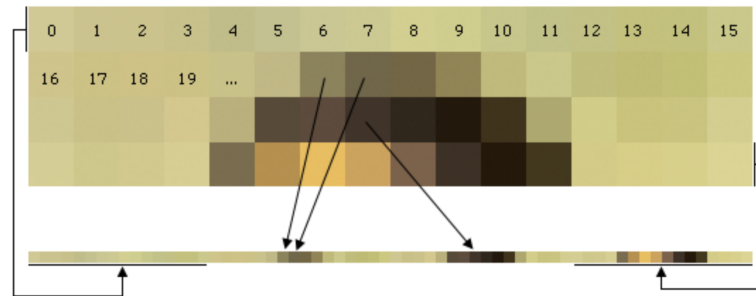


Figure 5.13: La liste de pixels de l'image lhoq.png.

Colorier les pixels dans une image

Nous pouvons aller encore plus loin dans l'utilisation du principe de listes appliqué à la description interne d'une image dans *Processing*. Il est possible en effet de créer nos propres variables de type `PImage` et d'assigner à ses pixels des couleurs, à l'aide de variables de type `color`. Les pixels de l'image sont accessibles au moyen d'un chiffre numéroté à partir de zéro, comme dans n'importe quelle liste. Dans cet exemple, on dessine une grille de trois pixels par trois. Nous allons dessiner cette image plus grande en l'étirant à 80 x 80 pixels afin de mieux la voir.

```
PImage img = createImage(3, 3, ARGB);
img.loadPixels();
img.pixels[0] = color(255, 0, 0);
img.pixels[1] = color(0, 255, 0);
```

```
img.pixels[2] = color(0, 0, 255);
img.pixels[3] = color(255, 0, 255);
img.pixels[4] = color(255, 255, 0);
img.pixels[5] = color(0, 255, 255);
img.pixels[6] = color(0, 0, 0);
img.pixels[7] = color(127, 127, 127, 255);
img.pixels[8] = color(255, 255, 255, 0);
img.updatePixels();
image(img, 10, 10, 80, 80);
```

Notez qu'il faut indiquer à *Processing* que nous allons modifier une image via `loadPixels()` et que nous avons terminé de modifier l'image via `updatePixels()`. En l'absence de ces instructions, on risque de ne pas voir les résultats de notre modification dans la liste. Notez également que vous pourriez aussi réaliser ce genre de peinture à numéro en utilisant des répétitions. Les applications créatives des listes sont multiples. Il ne vous reste plus qu'à laisser le champ libre à votre imagination.

5.5 Les méthodes

Une méthode est un bloc qui contient une série d'instructions que l'on souhaite réutiliser. L'intérêt des méthodes réside dans la possibilité de réutiliser du code : nous aimerions écrire une seule fois une action tout en pouvant la répéter autant de fois que nécessaire. En englobant notre code dans une méthode, il devient possible d'appeler celle-ci à différents moments de notre programme.

Vous connaissez déjà plusieurs méthodes, mais uniquement en tant qu'utilisateur. Par exemple, `rect()`, `ellipse()`, `line()` `stroke()` sont toutes des méthodes. En créant nos propres méthodes, nous nous rapprochons d'une certaine manière du rôle créatif des concepteurs de *Processing* : liberté nous est donnée d'inventer des formes ou des fonctionnalités qu'ils n'ont pas pu ou voulu écrire à l'origine.

Voici un exemple purement théorique de ce que la structuration d'un sketch à l'aide de méthodes pourrait éventuellement donner au final si nous devions dessiner un joli paysage.

```
background(255);
```

```
joliPaysage();
```

```
arbre(cypres, 0,300);
```

```
lune(400,100);
```

```
gazon(0,300,width,100);
```

```
mouton(50,133);
```

```
mouton(213,98);
```

```
mouton(155,88);
```

L'objectif, c'est de regrouper des éléments complexes du programme dans des mots clés que vous pouvez appeler autant de fois que vous le voulez, tout en les mélangeant avec les mots clés *Processing*. Ce procédé appelé encapsulation vous éloigne apparemment de votre code, mais c'est pour vous en donner un nouvel accès simplifié en le rendant davantage lisible. Cela permet également d'éviter les répétitions inutiles. Un peu plus d'ordre, pour un peu moins de copier-coller.

Mots clés

Lorsque nous créons nos propres méthodes, il faut donner à chacune d'entre elles un nom. Une fois la méthode définie, on peut s'en servir dans le programme. Il suffit de l'appeler par son nom.

Processing nous fournit déjà plusieurs méthodes que nous pouvons remplacer par nos propres versions. Ce sera le cas des méthodes `draw()`, `setup()`, `mousePressed()`... que vous découvrirez dans d'autres chapitres. Nous pouvons également créer des méthodes sur mesure en leur donnant le nom de notre choix. Dans ce cas, il faut simplement éviter d'utiliser un nom qui est déjà pris.

Décomposer

Jusqu'ici, nous avons programmé dans *Processing* directement, en commençant à saisir du code depuis le haut du programme et en laissant celui-ci s'exécuter jusqu'en bas. Lorsque nous voulons construire nos propres méthodes, nous devons commencer à décomposer nos programmes en plusieurs parties séparées. Cela nous permettra par exemple d'indiquer les parties qui

doivent s'exécuter tout de suite lorsque nous appuyons sur le bouton run de celles qui seront appelées par nos propres soins à l'intérieur du programme.

Par contre, en utilisant ce procédé de programmation, nous ne pourrons plus écrire des instructions directement dans *Processing* sans au préalable les avoir intégrées au sein d'une méthode ou d'une classe. Avec les méthodes, c'est tout ou rien.

void setup()

Processing nous offre une instruction ayant pour fonction de contenir le code de début de notre programme. Il s'agit de la méthode `setup()` :

```
void setup() {
}
```

C'est à l'intérieur des accolades de la méthode `setup()` que nous allons placer tout le code qui doit être exécuté au début de notre programme. Pour l'instant, ne cherchez pas à savoir à quoi sert le mot `void`, sachez juste qu'il faut l'écrire, suivi du mot `setup`, puis de parenthèses, et enfin les accolades. La plupart du temps, nous nous servons de la méthode `setup()` pour définir la taille de notre sketch. Cette taille ne peut être définie qu'une seule fois - ce qui tombe bien, car le démarrage n'a lieu qu'une seule fois dans le vie d'un programme.

```
void setup() {
    size(500,500);
}
```

Créer des méthodes sur mesure

Dans l'exemple qui suit, nous allons créer une méthode `dessinerMouton()` qui contient des instructions pour dessiner un mouton. En fait, ce mouton est caché dans une boîte, alors on ne voit qu'une boîte ! On y dessine également des trous, afin que le mouton puisse respirer. Nous appelons plusieurs fois cette méthode pour dessiner plusieurs moutons. Voici le code de ce dessin :

```
void setup() {
    size(600, 220);
    background(153,204,255);
    smooth();
    // l'appel à notre méthode de dessin d'un mouton
    dessinerMouton();
    translate(120, 60);
    dessinerMouton();
    translate(120, 60);
    dessinerMouton();
    translate(40, -60);
    dessinerMouton();
}

// la méthode pour dessiner le mouton
void dessinerMouton() {
    strokeWeight(3);
    strokeJoin(ROUND);
    stroke(0);
    fill(255);
```

```
rect(20, 40, 80, 40);
beginShape();
vertex(20, 40);
vertex(40, 20);
vertex(120, 20);
vertex(120, 40);
endShape(CLOSE);

beginShape();
vertex(100, 40);
vertex(120, 20);
vertex(120, 60);
vertex(100, 80);
endShape(CLOSE);

fill(0);
ellipse(40, 60, 5, 5);
ellipse(60, 60, 5, 5);
ellipse(80, 60, 5, 5);
}
```

Le début de ce programme se décrit à l'intérieur de la méthode `setup()`. En effet, puisque nous utilisons une méthode pour dessiner notre mouton, le reste du programme doit également être placé quelque part dans une méthode. En début du programme, nous allons donc saisir :

```
void setup() {
}
```

Ensuite, au sein des accolades de la méthode `setup()`, nous définissons la

taille de notre sketch et sa couleur de fond.

```
size(600, 220);
background(153,204,255);
```

Vous vous êtes peut-être posé la question du rôle de `smooth()` dans notre programme. Optionnelle, cette ligne de code permet toutefois d'améliorer le rendu des lignes en lissant leur tracé : elles apparaissent ainsi plus jolies sur l'écran de l'ordinateur. Enfin, nous dessinons notre mouton, en faisant appel à une méthode que nous avons définie plus bas dans le programme.

```
dessinerMouton();
```

Chaque fois que *Processing* tombe sur le mot `dessinerMouton()`, il vérifie si ce mot existe en tant que méthode quelque part dans le programme. Si cette méthode existe, il fait un détour par cette méthode et fait tout ce qui s'y trouve. S'il ne trouve pas cette méthode - et qu'elle n'existe pas ailleurs dans la liste des fonctionnalités proposées directement par *Processing* -, votre programme s'arrêtera avec une erreur d'exécution. Notez que vous pouvez écrire le mot clé `dessinerMouton()` autant de fois que vous voulez. Ici, dans ce programme, `dessinerMouton()` est écrit au final 4 fois :

```
dessinerMouton();
translate(120, 60);
dessinerMouton();
translate(120, 60);
dessinerMouton();
translate(140, -60);
```

```
dessinerMouton();
```

Nous avons placé entre chaque appel à la méthode `dessinerMouton()`, une instruction `translate(x,y)`. Cette instruction nous permet de ne pas dessiner quatre fois le même mouton au même endroit. Ce n'est pas le rôle de ce chapitre de vous expliquer les transformations comme `translate()`; sachez néanmoins que `translate()` sert à déplacer le point d'origine où débutera le tracé d'un dessin.

```
void dessinerMouton() {
  /* ... */
}
```

Enfin nous arrivons à notre méthode `dessinerMouton()` proprement dite. C'est ici que nous dessinons les lignes et formes nécessaires pour obtenir le tracé de notre animal. Nous ne commenterons pas cette partie, puisqu'il s'agit uniquement d'instructions que vous trouverez davantage décrites dans le chapitre sur les formes. Notez l'usage du mot-clé `void` devant le nom de notre méthode. Cela signifie qu'elle ne retourne rien. En faisant appel à elle, nous savons qu'elle n'a pas pour fonction de nous fournir des données.

La valeur de retour d'une méthode

Une méthode peut avoir une valeur de retour. Jusqu'ici, nous n'avons pas expérimenté cette particularité. Ni la méthode `setup()`, ni la méthode `draw()` ne retournent une valeur de retour. Le mot `void` a été placé devant chacune de ces deux méthodes pour bien préciser à *Processing* que rien ne doit être retourné lorsque l'on fera appel à elles.

L'emploi d'une méthode avec une valeur de retour suppose que nous cher-

chons à obtenir quelque chose d'elle en l'invoquant. Quand nous voulons savoir quelle heure est-il, nous demandons aux méthodes `second()`, `minute()`, ou `hour()` de nous donner en retour leurs valeurs sous forme d'un chiffre entier (`int`). Si ces méthodes ne nous donnaient rien (`void`) en retour, elles ne serviraient pas à grande chose.

Pour les méthodes qui doivent retourner une valeur à celle qui l'appelle, on indique un mot-clé avant pour indiquer le type de valeur qui doit être retourné. Une méthode dont le type est `int` nous retourne une valeur de type `int` correspondant à un nombre entier, une méthode dont le type est `float` nous retourne une valeur de type `float` (nombre à virgule), et ainsi de suite. Voici un exemple de méthode qui nous donne le nombre secondes depuis 00 :00 :00 ce matin.

```
int secondesAujourd'hui() {
  return hour() * 3600 + minute() * 60 + second();
}

void draw() {
  println( secondesAujourd'hui() );
}
```

Même si vous ne connaissez pas la méthode `draw()`, amusez-vous néanmoins à exécuter ce mini- programme et regardez les informations qui s'affichent dans la console située en bas de votre fenêtre d'édition de *Processing*. Vous verrez que la méthode `draw()` appelle en permanence la méthode `secondesAujourd'hui()` et utilise le résultat de cette méthode pour nous afficher les secondes.

Les paramètres d'une méthode

Une méthode peut accepter des paramètres. La plupart du temps, on les appelle des arguments. Ces paramètres doivent avoir chacun un type et un nom, tout comme les variables. Pour appeler une méthode, on écrit son nom, et on le fait suivre d'une parenthèse ouvrante et d'une autre fermante. Entre ces parenthèses, on place les paramètres de la méthode. Ce qu'on y met sera envoyé dans la méthode.

```
multiplier(2, 2);
```

Une fois arrivé dans le corps de la méthode, *Processing* peut accéder à leur valeur, comme il le fait avec les variables. Ainsi dans l'exemple ci-après, lorsque cette méthode est appelée avec les arguments 2 et 2, la valeur de *a* = 2, et celle de *b* = 2 également. La valeur de retour de cette méthode sera donc 4 (2 fois 2 égale 4).

```
int multiplier(int a, int b) {  
    return a * b;  
}
```

Notez que c'est la position des arguments qui détermine quelle valeur sera affectée à quel argument.

Pour résumer notre exemple, on a créé une méthode qui retourne le résultat de la multiplication de ses deux arguments. Un commentaire précède la définition de la méthode (une bonne pratique de programmation pour se rappeler ultérieurement de la fonction d'un morceau de code).

```
/* Retourne le résultat de la multiplication */
```

```
/* de ses deux arguments. */  
int multiplier(int a, int b) {  
    return a * b;  
}  
  
void setup() {  
    int resultat = multiplier(2, 2);  
    print(resultat);  
}
```

La console de *Processing* affichera : "4".

La portée des variables

Profitions de ce chapitre sur les méthodes et les variables pour vous mettre en garde contre une erreur classique qui peut survenir lorsque l'on utilise des variables et des méthodes dans un programme. Les variables - que ce soit des objets ou des types fondamentaux de données - ne sont pas forcément accessibles à l'ensemble de votre programme! Tout dépend de l'endroit où elles ont été déclarées. Une variable déclarée à l'intérieur d'une méthode ne sera accessible que dans celle-ci :

```
void setup() {  
    int x = 10;  
}  
  
void draw() {  
    /* Le programme générera une erreur */  
    /* car la variable x n'existe qu'à l'intérieur */  
    /* de la méthode setup() */
```

```
    x = x + 1;
}
```

En lançant l'exécution du programme ci-dessus, la console de *Processing* affichera le message d'erreur suivant : "The field Component.x is not visible". Pour qu'une variable soit accessible à tout votre programme il faut la déclarer en en-tête comme ceci :

```
int x;

void setup() {
    x = 10;
}

void draw() {
    x = x + 1;
}
```

5.6 Les objets

La **programmation orientée objet** (POO) permet de structurer son programme à partir des éléments **concrets** présents dans l'espace de dessin (balles, murs, personnages, etc.). Un objet est un modèle qui peut être dupliqué et dont **chaque copie est unique**. Ces 2 notions constituent les briques de base d'une application structurée. Un objet est composé de **caractéristiques** (propriétés) et d'**actions** (méthodes). Chaque **instance** (chaque copie unique) d'un objet possède sa vie propre avec des caractéristiques spécifiques tout en pouvant effectuer potentiellement les mêmes actions que ses sœurs (les autres instances du même objet). Ne vous inquiétez pas si ces définitions vous paraissent obscures à première lecture. Au fil de ce chapitre, nous allons détailler ces notions à l'aide d'un exemple simple et concret : le tracé d'une puis de deux balles (de simples cercles) à l'écran. Pour réaliser ce dessin, nous faisons appel à un modèle de balle (l'objet) et à ses copies (les instances) qui possèdent chacune des caractéristiques différentes.

Le modèle de la balle est composé des **caractéristiques** (variables) suivantes :

- position sur l'axe x
- position sur l'axe y
- couleur

Le modèle de balle contiendra l'action (méthode) suivante :

- afficher

Quand créer un objet

Comment faire pour dessiner plusieurs balles à l'écran ? La solution la plus évidente semble être de dupliquer les variables qui caractérisent la balle et les instructions permettant de l'afficher et de la gérer. La taille du code sera

proportionnelle au nombre d'éléments visibles à l'écran. Plus j'ai de balles plus mon code sera long.

Ce procédé pose deux problèmes :

- si l'on veut modifier ou ajouter des actions à l'ensemble de ces balles, on devra modifier autant de fois le code qu'il y a d'éléments affichés,
- au fur et à mesure que l'on rajoute des éléments à l'écran le programme s'allonge au point de devenir ingérable (et si notre programme comportait 1'000'000 de balles ?).

Pour pallier à ces limitations, nous allons transformer notre balle en **objet**. Dès qu'une entité (balle, avatar, forme, etc.) de votre programme devient trop complexe ou qu'elle doit exister en **plusieurs exemplaires**, il faut en faire un objet.

Créer un objet

La création d'un objet se passe en deux étapes : la définition du modèle de l'**objet** et la création d'une **copie unique** de l'objet (instance).

Le modèle *Processing* utilise le mot clé `class` pour définir un objet. Sa syntaxe s'apparente à la définition d'une méthode : `class nomObjet`. Toutes les **caractéristiques** et les **actions** sont écrites à l'intérieur. En général on va écrire la définition d'un **objet** tout à la fin de notre code. éventuellement si l'objet est complexe, on créera un nouvel onglet dans la fenêtre d'édition de *Processing* (un nouveau fichier) afin de le séparer du reste du code.

```
class Balle
```

L'instance Une fois le modèle défini, il faut créer une copie de ce **modèle** qui sera unique. *Processing* utilise le mot clé `new` pour créer une **instance** d'un **objet** : `new nomObjet()` ;. Il faut stocker cet **objet** dans une **variable** afin de

pouvoir le manipuler ultérieurement.

```
Balle maBalle = new Balle();
```

Dans l'exemple ci-dessus, nous déclarons une **variable** `maBalle` et nous lui assignons une copie de l'objet `Balle`. `maBalle` fait référence à cet **objet** et permet d'agir sur lui dans la suite du programme. Afin de rendre l'objet disponible dans tout le programme, nous plaçons la déclaration en en-tête de l'application.

Les caractéristiques

Les **objets** ont des **caractéristiques** qui les définissent et les rendent uniques. Ce sont des **variables** qui sont déclarées au début de l'**objet**.

```
class Balle {
    //Déclaration des paramètres de base de la balle
    float x;
    float y;
    color couleur;
}
```

Pour modifier une **caractéristique** après la création de l'**objet**, il faut procéder de la manière suivante : `nomDeLInstance.nomDeLaCaractéristique = valeur`; . Par exemple :

```
maBalle.x = 100;
```

Le constructeur

Le **constructeur** est une **méthode** appelée lorsque l'**objet** est créé. Il est l'équivalent de la **méthode** `setup()` de l'application. Il porte toujours le nom de l'**objet**. Le constructeur va prendre un certain nombre de variables en **paramètre** et les assigner à l'objet :

```
class Balle {
    //Déclaration des paramètres de base de la balle
    float x;
    float y;
    color couleur;
}

Balle (float nouvX, float nouvY, color nouvCouleur) {
    x = nouvX;
    y = nouvY;
    couleur = nouvCouleur;
}
```

Lorsque nous allons créer l'instance (une copie) de l'**objet**, nous allons directement lui donner ses caractéristiques propres en paramètre. L'exemple ci-dessous crée une balle blanche placée aux coordonnées 100, 100 de la fenêtre de visualisation.

```
maBalle = new Balle(100, 100, color(255));
```

Attention ! Nous venons uniquement de créer une copie du modèle de l'**objet**. Il n'est pas encore affiché à l'écran.

Les actions

Les actions d'un **objet** représentent les différentes choses qu'il peut effectuer. Ce sont des **méthodes** mentionnées (déclarées) à l'intérieur de l'**objet**. Appeler une action sur une instance d'un **objet** se fait de la manière suivante : `nomDeLInstance.nomDeLaMethode()` ; Dans notre exemple, notre balle comportera une seule **action** : être affichée. Nous allons utiliser les instructions `fill()` et `ellipse()` pour la dessiner.

```
class Balle {
  //Déclaration des paramètres de base de la balle
  float x;
  float y;
  color couleur;

  Ball (float nouvX, float nouvY, color nouvCouleur) {
    x = nouvX;
    y = nouvY;
    couleur = nouvCouleur;
  }

  void display() {
    fill(couleur);
    ellipse(x, y, 40, 40);
  }
}
```

Pour afficher notre balle, il faut appeler sa méthode `display()` depuis sa copie dans la méthode `draw()` du programme.

```
void draw() {
  maBalle.display();
}
```

Programme final

```
//Déclaration et création d'une instance de l'objet Balle
Balle maBalle = new Balle(100, 100, color(255));
```

```
void setup() {
  smooth(); //Lissage des dessins
  size(400, 200); //Taille de la fenêtre
}
```

```
void draw() {
  background(0); //On dessine un fond noir
  noStroke(); //On supprime le contour
  maBalle.display(); //Affichage de la balle
}
```

```
class Balle {
  //Déclaration des paramètres de base de la balle
  float x;
  float y;
  color couleur;

  //Constructeur de la balle
  Ball (float nouvX, float nouvY, color nouvCouleur) {
```

```

    x = nouvX;
    y = nouvY;
    couleur = nouvCouleur;
}

//Dessin de la balle
void display() {
    fill(couleur);
    ellipse(x, y, 40, 40);
}
}

```

Multiplier les balles

Créer une seconde balle devient, grâce au système des objets, un jeu d'enfant. Il suffit de déclarer une deuxième balle, par exemple `maBalle2` et de l'afficher.

```

//Déclaration et création de plusieurs instances de l'objet
Balle Balle maBalle1 = new Balle(100, 100, color(255));
Balle maBalle2 = new Balle(200, 100, color(128));

void setup() {
    smooth(); //Lissage des dessins
    size(400, 200); //Taille de la fenêtre
}

void draw() {
    background(0); //On dessine un fond noir

```

```

    noStroke(); //On supprime le contour

    maBalle1.display(); //Affichage de la balle 1
    maBalle2.display(); //Affichage de la balle 2
}

class Balle {
    //Déclaration des paramètres de base de la balle
    float x;
    float y;
    color couleur;

    //Constructeur de la balle
    Balle(float nouvX, float nouvY, color nouvCouleur) {
        x = nouvX;
        y = nouvY;
        couleur = nouvCouleur;
    }

    //Dessin de la balle
    void display() {
        fill(couleur);
        ellipse(x, y, 40, 40);
    }
}

```

N'hésitez pas à vous approprier ce programme en ajoutant autant de balles que vous voulez.

5.7 Les commentaires

Commenter et documenter son programme sont essentiels pour maintenir un code clair et pour faciliter la collaboration avec d'autres personnes. Cette pratique de programmation permet également de se rappeler ultérieurement l'utilité de telle ou telle variable, méthode, etc. Au moment de la conception d'un programme, on possède une vue d'ensemble du code et parfois commenter notre sketch peut nous sembler inutile. Pourtant, lorsqu'on veut retravailler ou réutiliser ce code quelques jours, voire quelques mois plus tard, l'usage de commentaires nous permet de nous y replonger plus rapidement. *Processing* offre deux manières distinctes de commenter son code : les commentaires en ligne, et les blocs de commentaires multi-lignes.

Commentaires en ligne

Pour écrire un commentaire qui ne prend qu'une seule ligne, il suffit de placer les caractères `//` au début du commentaire. Tout le texte qui est écrit à sa droite sur la même ligne sera considéré comme commentaire et donc ne sera pas pris en compte par le programme lors de son exécution. Exemples de commentaires en ligne :

```
void setup() {  
  // Définit la taille du programme  
  size(400, 300);  
  smooth(); // Active le lissage des contours  
}
```

Blocs de commentaires

Si une explication nécessite un commentaire plus long, on peut l'écrire sur plusieurs lignes en le plaçant entre les caractères `/*` et `*/`. Par exemple :
`/* La méthode setup initialise le programme, on peut y définir la
taille de la fenêtre, définir l'état initial du programme, etc. */
void setup() {
}`

On peut commenter un bloc de ligne en allant dans le menu edit de *Processing* puis en cliquant sur comment/uncomment. On peut procéder de même pour enlever le commentaire sur un bloc.

Utilisation judicieuse des commentaires

Commenter un programme ne signifie pas qu'il faille écrire le but de chaque ligne ! De manière générale, on inscrira un bloc de commentaires avant les méthodes pour expliquer leur utilité et éventuellement la façon de s'en servir. On évitera de commenter les méthodes de base de *Processing* telles que `setup()`, `draw()`, etc. On ne commentera pas non plus les instructions génériques telles que `size()`, `fill()`, `ellipse()`, etc. Leur utilité est évidente dès lors que vous êtes relativement familiarisé avec *Processing*.

Lorsqu'un ensemble de lignes effectue une action commune (par exemple modifier deux variables `x` et `y`), on évitera de mettre un commentaire pour chaque ligne et on utilisera de préférence un bloc de commentaires. Par exemple :

```
x = x + 10; //Ajout de 10 à la coordonnée x  
y = y + 10; //Ajout de 10 à la coordonnée y
```

pourrait être écrit :

```
x = x + 10;  
y = y + 10;
```

```
//Modification des coordonnées x et y
```

Animer avec *Processing*

Sommaire

6.1 La méthode draw	75
6.2 La ligne de temps	80
6.3 L'animation d'un objet	83
6.4 L'animation de plusieurs objets	88
6.5 La lecture du son	93

Jusqu'ici, nous avons créé ce que nous pouvons appeler des programmes linéaires : on démarre le programme, celui-ci exécute notre dessin, et tout s'arrête à la fin des instructions de notre sketch, en bas de la fenêtre d'édition. Cependant, *Processing* n'est pas uniquement un environnement de dessin écrit, c'est également un environnement interactif.

6.1 La méthode draw

Pour qu'il y ait interactivité, il nous faut du temps, en d'autres termes, un moyen de prolonger notre dessin pour que celui-ci puisse se modifier en suivant chronologiquement certaines étapes, selon différents facteurs et conditions.

C'est le rôle de la boucle infinie, appelée en permanence par la machine pour réactualiser notre dessin. Dans *Processing*, cette boucle infinie s'écrit `draw()`. Elle est souvent accompagnée par la méthode `setup()` qui permettra de préparer la fenêtre de visualisation (l'espace de dessin), par exemple en lui donnant une taille au départ. C'est l'absence de ces deux méthodes qui rend *Processing* inerte. Si votre code ne dispose pas de méthode `draw()`, l'exécution du programme s'arrêtera à la fin du code de votre programme.

draw()

Commencez un nouveau programme *Processing* vide, tapez les lignes suivantes dans la fenêtre d'écriture et appuyez sur le bouton run :

```
void draw() {  
  background( random(255) );  
}
```

Ça y est, vous avez activé l'animation dans *Processing*. Vous devriez maintenant voir une fenêtre qui clignote 30 fois par seconde avec une couleur grise aléatoire quelque part entre le noir et le blanc. C'est la méthode `random(255)` qui donne en retour une valeur aléatoire entre 0 et 255. C'est cette valeur qui est ensuite récupérée par les parenthèses de la méthode `background()` et appliquée sur le fond de la fenêtre de visualisation de l'espace de dessin. Comme le tout se passe de manière répétée, on a l'impression d'assister à une animation.

Par défaut, les instructions qui se trouvent entre les deux accolades de la méthode `draw()` seront appelées 30 fois par seconde. 30 fois par seconde *Processing* ouvrira cette méthode et regardera ce qui est écrit dedans pour l'exécuter. Nous pouvons mettre autant d'instructions que nous voulons à l'intérieur de cette méthode. Ces instructions seront jouées par *Processing* de manière cyclique, tel un métronome.

Attention aux erreurs d'écriture, car dès que *Processing* en rencontrera une, votre programme s'arrêtera net - c'est la fin de votre animation. En revanche, si vous avez écrit une méthode `draw()` sans erreurs, elle sera appelée en boucle, 30 fois par seconde, jusqu'à ce que l'utilisateur arrête le programme, qu'une panne d'électricité arrive, ou que la fin du monde se produise. C'est donc grâce à cette méthode exécutant du code en répétition que nous allons pouvoir créer des animations.

Fréquence d'exécution

Il est possible de spécifier une valeur différente à notre métronome en utilisant la méthode `frameRate()`. Si vous modifiez l'exemple précédent comme suit, vous remarquerez que la vitesse de l'animation a été diminuée (divisée par 3).

```
void draw() {  
  frameRate(10);  
  background( random(255) );  
}
```

Nombre de fois où `draw()` a été appelée

Processing peut aussi compter le nombre de fois que cette méthode `draw()` a été appelée, depuis le lancement du programme via la variable `frameCount`.

`setup()`

Souvent, voire la plupart du temps, il est nécessaire de placer certaines instructions au tout début du programme. C'est la nature et la portée de ces instructions sur le programme qui nous incitent à les placer à cet endroit. Par exemple, dans *Processing*, la taille de la fenêtre de visualisation du dessin ne peut être définie qu'une seule fois dans un sketch ; ce paramétrage de l'espace de dessin ne peut donc être placé à l'intérieur de la méthode `draw()` car cette méthode s'exécute plusieurs fois durant le déroulement du programme.

A moins de se satisfaire de la dimension par défaut de 100x100 pixels, définir la taille de la fenêtre de visualisation peut s'avérer très utile. C'est pour toutes ces raisons qu'une deuxième méthode complémentaire a été créée :

la méthode `setup()`. Les subtilités des différents signes qui composent cette méthode sont pour l'instant sans importance. Sachez simplement qu'il faut écrire `void setup()` tout au début, suivi d'une ouverture d'accolades, saisir ensuite les instructions que vous voulez exécuter, et enfin terminer avec la fermeture des accolades.

```
void setup() {  
  size(500, 500);  
}  
void draw() {  
  background(random(255));  
}
```

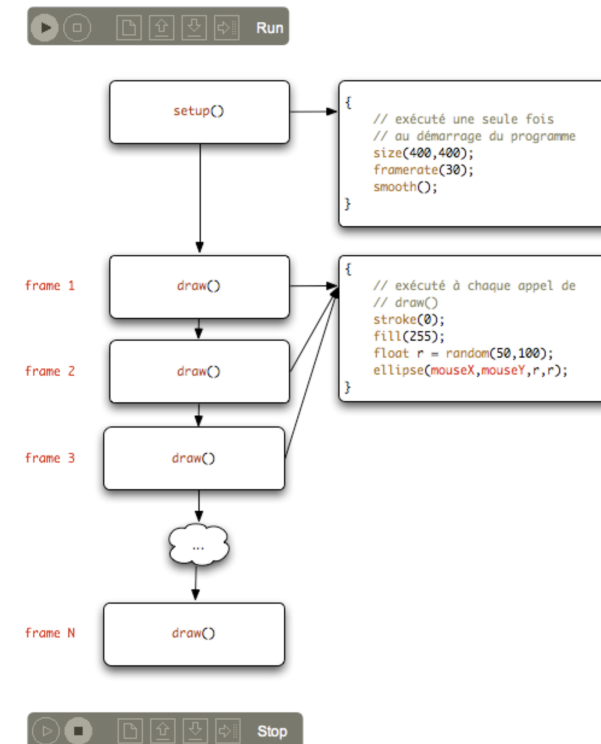


Figure 6.1: Schéma du principe du traitement d'un sketch dans Processing.

Lorsque ce sketch est lancé, *Processing* exécute tout d'abord les instructions qui se trouvent à l'intérieur de la méthode `setup()`. Ensuite, la méthode `draw()` commencera à être appelée de manière répétée, telle un métronome. Ce principe de fonctionnement est illustré par le schéma ci-dessus.

Background()

Voici un programme qui remplit progressivement l'écran avec des ellipses.

```
void setup() {  
  size(200, 200);  
  fill(0);  
}  
  
void draw() {  
  ellipse(random(200), random(200), 20, 20);  
}
```

Comme vous pouvez le constater dans les captures d'écran, cette animation finira par remplir notre espace de dessin complètement par du noir. C'est peut-être l'œuvre conceptuelle ultime de toute notre carrière artistique ; néanmoins il serait bien utile d'apprendre à animer une seule ellipse, la voir évoluer dans l'espace ou changer de forme, sans que ses positions antérieures soient affichées en même temps. Dans ce cas, il suffit d'ajouter un nettoyage de fond de l'espace de dessin à l'aide de la méthode `background()`. Celle-ci prend une à trois valeurs de couleur, comme pour la méthode `fill()` ou `stroke()`. Voici une modification du programme qui permettra d'afficher maintenant une seule forme animée :

```
void setup() {  
  size(200, 200);  
  fill(0);  
}
```

```
void draw() {  
  background(255);  
  ellipse( 100, 100, random(100), random(100));  
}
```

En réalité, ce que nous demandons au programme, c'est 30 fois par seconde d'effacer l'intégralité de notre dessin et de tracer une nouvelle forme par dessus avec un nouveau fond blanc.

Ajouter un fondu

Il existe une astuce, souvent utilisée dans la communauté des utilisateurs de *Processing*, qui consiste à effacer le fond de l'image à l'aide d'un rectangle semi-transparent plutôt qu'avec l'instruction `background()`. Ce procédé permet d'obtenir un effet d'effacement graduel, de fondu.

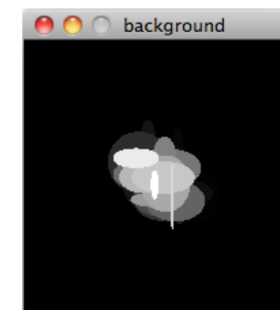


Figure 6.2: Ajouter un effet d'effacement graduel.

```
void setup() {  
  size(200,200);  
  background(0);  
}
```

```
noStroke(); }
```

```
void draw() {  
  fill(0, 0, 0, 20);  
  rect(0, 0, 200, 200);  
  fill(255);  
  ellipse(100 + random(-20,20), 100 + random(-20,20), random(50),  
random(50));  
}
```

6.2 La ligne de temps

Pour créer une animation, il faut qu'il y ait du mouvement. Le mouvement implique un changement du dessin dans le temps, par exemple une modification de la position ou de la couleur d'un de ces éléments graphiques. Les informations liées à ces changements peuvent être stockées dans des variables. Pour créer des animations, il faut savoir à quel moment nous nous trouvons par rapport à une ligne de temps. Pour ce faire, nous pouvons soit utiliser l'heure qu'il est, soit compter (par exemple de un à dix).

Quelle heure est-il ?

Nous allons créer une horloge en appelant les méthodes `hour()`, `minute()` et `second()` de *Processing*. Nous allons utiliser le résultat de l'appel de ces méthodes pour faire varier la position sur l'axe horizontal de trois minces rectangles. Le code pour réaliser cette horloge est très simple :

```
void setup() {
  size(60, 60);
  noStroke();
}

void draw() {
  background(0);
  // Les heures vont de 0 à 23,
  // nous les convertissons à une échelle de 0 à 60
  rect((hour() / 24.0) * 60, 0, 1, 20);
  rect(minute(), 20, 1, 20);
  rect(second(), 40, 1, 20);
}
```

```
}
```

Les images sont dessinées les unes par-dessus les autres. L'appel à l'instruction `background(0)` remplit l'espace de dessin avec la couleur définie en paramètre (le nombre 0 correspondant à du noir) ce qui revient à chaque fois à effacer l'image précédente.

Mesurer le temps qui passe

On peut obtenir le temps qui s'est écoulé depuis le début de l'exécution d'un sketch en utilisant la méthode `millis()`. Celle-ci retourne un nombre en millisecondes. Cette valeur peut être exploitée pour créer des animations. Il y a mille millisecondes dans une seconde : une précision suffisante pour animer des formes ou des sons.

Par ailleurs, vous pouvez également avoir besoin de créer des animations cycliques dont la fréquence (le nombre d'animations par cycle) s'adapte en fonction d'autres éléments, par exemple la taille de la fenêtre de visualisation de l'espace de dessin. Pour déterminer à partir d'une valeur donnée ce nombre d'animations par cycle, la solution la plus simple consiste à utiliser l'opérateur modulo `%`. En mathématique, le modulo permet d'obtenir le reste d'une division par un nombre. Par exemple, l'expression `println(109 % 10);` affichera 9, car 109 divisé par 10, donne 10 comme quotient et 9 comme reste. De manière plus générale, si on prend deux nombres `x` et `y`, le reste de la division de `x` par `y` est strictement inférieur à `y`. L'opérateur modulo nous permet donc de compter sans jamais dépasser un certain nombre (la base du modulo).

Dans le prochain exemple, nous allons utiliser la méthode et l'opérateur `millis()` pour dessiner un cercle qui parcourt très rapidement le sketch selon des trajectoires diagonales. Ce cercle laisse derrière lui une traînée qui s'efface graduellement. Le code de cette animation est le suivant :

```
void setup() {
  size(320, 240);
  noStroke();
  frameRate(60);
  smooth();
}

void draw() {
  fill(0, 0, 0, 10);
  rect(0, 0, width, height);
  fill(255);
  ellipse(millis() % width, millis() % height, 20, 20);
}
```

L'effacement du dessin précédent s'effectue à l'aide d'un rectangle semi-transparent. Ce procédé permet d'obtenir un effet de flou de mouvement contrairement à l'instruction `background()`. La présence de l'instruction `smooth()` au début du programme s'explique par notre amour des courbes lisses. Un conseil : basez vos animations sur la mesure du temps plutôt que sur le décompte du nombre d'images dessinées. La vitesse de rendu peut très sensiblement se dégrader lorsque les capacités de l'ordinateur sont occupées à gérer un compteur : les animations deviennent alors moins fluides, plus saccadées. Ces limitations de performance n'existent pas lorsque l'on utilise les instructions associées à l'horloge interne de l'ordinateur pour mesurer le temps qui passe.

Pour satisfaire votre curiosité et malgré notre mise en garde sur l'utilisation de la méthode du décomptage du nombre d'images dessinées, nous allons

néanmoins voir comment créer des animations à l'aide de compteurs, le principe de mis en oeuvre étant très simple.

Animer à l'aide d'un compteur

Nous allons maintenant voir comment créer une animation en comptant, même si, rappelons-le, nous vous déconseillons cette méthode pour des questions de performance. Dans notre exemple, l'animation représentant une simple ligne qui tourne sans cesse. Le code est relativement simple, le compteur d'images constituant l'essentiel du programme. A chaque degré de rotation de la ligne correspond le numéro d'une image dessinée.

```
int compteur;
void setup() {
  size(320, 240);
  frameRate(60);
  fill(0, 0, 0, 10);
  stroke(255);
  smooth();
  compteur = 0;
}

void draw() {
  compteur = compteur + 1;

  rect(0, 0, width, height);
  translate(width / 2, height / 2);
  rotate(radians(compteur));
  line(-height, -height, height, height);
}
```

```
}
```

On déclare un compteur en en-tête de l'application.

```
int compteur;
```

Dans le `setup()` on initialise notre compteur.

```
void setup() {  
    size(320, 240);  
    frameRate(60);  
    fill(0, 0, 0, 10);  
    stroke(255);  
    smooth();  
    compteur = 0;  
}
```

A chaque appel de méthode `draw()` on incrémente notre compte de 1.

```
compteur = compteur + 1;
```

On spécifie ensuite le repère de l'espace de dessin pour dessiner au centre et on génère une rotation qui dépend de `compteur`.

```
translate(width / 2, height / 2);  
rotate(radians(compteur));
```

Une traîne est visible dans l'animation. La superposition des lignes en s'effaçant graduellement donne un effet moiré.

6.3 L'animation d'un objet

En combinant animation et objet, il devient possible de concevoir des applications plus ambitieuses. Animer un objet revient à ajouter à son modèle des actions de type : se déplacer, rebondir, tester les collisions, etc. Dans ce chapitre, nous allons apprendre comment animer une balle et la faire rebondir sur les quatre bords de l'écran. Le résultat final se présentera comme ci-dessous. Afin de mieux visualiser la trajectoire de la balle, nous avons intégré dans l'animation un effet de traîne.

Code de base

Comme point de départ, nous allons reprendre le programme de la balle du chapitre "Les objets" qui nous permettait d'afficher une balle à l'écran et lui ajouter progressivement des morceaux de code. Les nouvelles parties du sketch sont signalées avec la mention *//AJOUT* ou *//DEBUT AJOUT* et *//FIN AJOUT*. Pour vous familiariser avec le fonctionnement de ce nouveau programme, vous pouvez également copier tout le bloc du code initial dans votre fenêtre d'édition de *Processing* et progressivement lui ajouter la classe ou la méthode concernée.

```
//Déclaration et création d'une instance de l'objet Balle
Balle maBalle = new Balle(100, 100, color(255));
```

```
void setup() {
  smooth(); //Lissage des dessins
  size(400, 200); //Taille de la fenêtre
}
```

```
void draw() {
  background(0); //On dessine un fond noir
  noStroke(); //On supprime le contour
  maBalle.display(); //Affichage de la balle
}

class Balle {
  //Déclaration des paramètres de base de la balle
  float x;
  float y;
  color couleur;

  //Constructeur de la balle
  Balle (float nouvX, float nouvY, color nouvCouleur) {
    x = nouvX;
    y = nouvY;
    couleur = nouvCouleur;
  }

  //Dessin de la balle
  void display() {
    fill(couleur);
    ellipse(x, y, 40, 40);
  }
}
```

Le déplacement

La balle doit pouvoir se déplacer sur les axes x et y. Nous allons créer deux variables dans le modèle de l'objet qui caractériseront sa vitesse sur les axes x et y. Ensuite nous allons ajouter une nouvelle méthode bouge() dans le modèle de l'objet qui sera appelée depuis la méthode draw() du programme. Cette méthode va, à chaque fois que l'objet est affiché, modifier la position de la balle par rapport à sa vitesse. Il faudra aussi initialiser les variables décrivant la vitesse dans le constructeur. Pour commencer, nous allons leur donner une valeur fixe.

```
class Balle {
  //Déclaration des caractéristiques de base de la balle
  float x;
  float y;
  float vitesseX; //AJOUT
  float vitesseY; //AJOUT color couleur;

  //Constructeur de la balle
  Balle (float nouvX, float nouvY, color nouvCouleur) {
    x = nouvX;
    y = nouvY;
    couleur = nouvCouleur;
    vitesseX = 2; //AJOUT
    vitessesY = 2; //AJOUT
  }

  //Dessin de la balle void display() {
    fill(couleur);
```

```
    ellipse(x, y, 40, 40);
  }

  //DEBUT AJOUT
  void bouge() {
    x = x + vitesseX; y = y + vitesseY;
  }
  //FIN AJOUT
}
```

Il faudra ensuite appeler la méthode bouge() depuis la méthode draw(). Par ailleurs, nous ajoutons un appel à l'instruction background() pour effacer l'écran à chaque nouvelle image.

```
void draw() {
  background(0); //On dessine un fond noir
  noStroke(); //On supprime le contour

  //Déplacement et affichage de la balle
  maBalle.bouge(); //AJOUT
  maBalle.display();
}
```

Les collisions

Pour le moment, dès que la balle touche le bord de l'écran, elle continue son chemin. Selon l'exemple de la balle qui rebondit sur les coins de l'écran du chapitre "La ligne de temps", nous allons ajouter une méthode testCollision

qui inversera la vitesse de la balle lorsqu'elle touche les bords de l'écran.

```
class Balle {
    //Déclaration des caractéristiques de base de la balle
    float x;
    float y;
    float vitesseX;
    float vitesseY;
    color couleur;

    //Constructeur de la balle
    Balle (float nouvX, float nouvY, color nouvCouleur) {
        x = nouvX;
        y = nouvY;
        couleur = nouvCouleur;
        vitesseX = 2;
        vitesseY = 2;
    }

    //Dessin de la balle
    void display() {
        fill(couleur);
        ellipse(x, y, 40, 40);
    }

    void move() {
        x = x + vitesseX;
        y = y + vitesseY;
```

```
    }

    //DEBUT AJOUT
    void testCollision() {
        //Si la balle touche un mur, elle rebondit
        if (x > width-20 || x < 20) {
            vitesseX = vitesseX *-1;
        }
        if (y > height-20 || y < 20) {
            vitesseY = vitesseY * -1;
        }
    }
    //FIN AJOUT
}
```

Il faut ensuite appeler la méthode testCollision() depuis la méthode draw().

```
//ON REMPLACE L'INSTRUCTION BACKGROUND() PAR CES DEUX LIGNES
fill(0, 0, 0, 1); // Couleur avec transparence.
rect(0, 0, width, height);
noStroke();

//Déplacement et affichage de la balle
maBalle.bouge();
maBalle.testCollision();//AJOUT
maBalle.display();
```

Code Final

Voici le code final, une fois toutes ces modifications effectuées.

```
//Déclaration et création d'une instance de l'objet Balle
Balle maBalle = new Balle(100, 100, color(255));

void setup() {
  smooth(); //Lissage des dessins
  size(400, 200); //Taille de la fenêtre
}

void draw() {
  fill(0, 0, 0, 1);
  rect(0, 0, width, height);
  noStroke();

  //Déplacement et affichage de la balle
  maBalle.bouge();
  maBalle.testCollision();
  maBalle.display();
}

class Balle {
  //Déclaration des caractéristiques de base de la balle
  float x;
  float y;
  float vitesseX;
  float vitesseY;
```

```
color couleur;

//Constructeur de la balle
Balle (float nouvX, float nouvY, color nouvCouleur) {
  x = nouvX;
  y = nouvY;
  couleur = nouvCouleur;
  vitesseX = 2;
  vitesseY = 2;
}

//Dessin de la balle
void display() {
  fill(couleur);
  ellipse(x, y, 40, 40);
}

void bouge() {
  x = x + vitesseX;
  y = y + vitesseY;
}

void testCollision() {
  //Si la balle touche un mur, elle rebondit
  if (x > width-20 || x < 20) {
    vitesseX = vitesseX *-1;
  }
  if (y > height-20 || y < 20) {
```

```
        vitesseY = vitesseY * -1;  
    }  
}  
}
```

N'hésitez pas à modifier certains paramètres du programme pour vous approprier davantage son fonctionnement.

6.4 L'animation de plusieurs objets

Dès lors qu'un objet est créé dans un programme, il est possible de le multiplier facilement et rapidement. Deux solutions sont possibles :

- Dans le chapitre d'introduction aux objets, nous avons vu qu'on pouvait obtenir deux balles à l'écran en déclarant une seconde copie de la balle et en l'affichant à son tour dans la méthode `draw()`. Ce procédé devient toutefois lourd lorsque le nombre d'objets à reproduire est supérieur à deux.
- Lorsqu'il y a plus de deux objets à animer, il est préférable d'utiliser des listes. Petit rappel du chapitre consacré à cette notion : les listes permettent de gérer facilement un ensemble d'éléments semblables que ce soit des chiffres, des images et même des objets.

Dans ce chapitre, nous allons poursuivre l'exemple de la balle rebondissante. Nous allons ajouter plusieurs balles en utilisant des listes puis ajouter une méthode permettant de gérer les collisions entre les balles.

Code de base

Comme point de départ, nous allons reprendre le code de la balle du chapitre "Animer un objet" qui nous permettait d'afficher une balle rebondissante à l'écran. Tout au long du chapitre, nous allons ajouter des portions de code à l'exemple de base. Les nouvelles parties sont signalées avec la mention `//AJOUT` ou `//DEBUT AJOUT` et `//FIN AJOUT`. Pour vous familiariser avec le fonctionnement de ce nouveau programme, vous pouvez également copier tout le bloc du code initial dans votre fenêtre d'édition de *Processing* et progressivement lui ajouter la classe ou la méthode concernée.

`//Déclaration et création d'une instance de l'objet Balle`

```
Balle maBalle = new Balle(100, 100, color(255));
```

```
void setup() {
    smooth(); //Lissage des dessins
    size(400, 200); //Taille de la fenêtre
}
```

```
void draw() {
    fill(0, 0, 0, 1);
    rect(0, 0, width, height);
    noStroke();

    //Déplacement et affichage de la balle
    maBalle.bouge();
    maBalle.testCollision();
    maBalle.display();
}
```

```
class Balle {
    //Déclaration des caractéristiques de base de la balle
    float x;
    float y;
    float vitesseX;
    float vitesseY;
    color couleur;

    //Constructeur de la balle
    Balle(float nouvX, float nouvY, color nouvCouleur) {
```

```
x = nouvX;
y = nouvY;
couleur = nouvCouleur;
vitesseX = 2;
vitesseY = 2;
}

//Dessin de la balle
void display() {
    fill(couleur);
    ellipse(x, y, 40, 40);
}

void bouge() {
    x = x + vitesseX;
    y = y + vitesseY;
}

void testCollision() {
    //Si la balle touche un mur, elle rebondit
    if (x > width-20 || x < 20) {
        vitesseX = vitesseX * -1;
    }
    if (y > height-20 || y < 20) {
        vitesseY = vitesseY * -1;
    }
}
}
```

Lister les balles

Nous allons maintenant appliquer le concept de listes à notre balle rebondissante. Cela nous permettra d'avoir plusieurs balles à l'écran, sans dupliquer le code! En premier lieu, nous allons déclarer une liste de balles et non plus une seule balle. Pour cela, nous allons utiliser une variable `nbreBalle` pour stocker le nombre de balles utilisées dans le programme. Nous allons remplacer la déclaration en en-tête suivant

```
Balle maBalle = new Balle(100, 100, color(255));
par
```

```
//Déclaration d'une variable contenant le nombre de balles
int nbreBalle = 3;
```

```
//Déclaration d'une liste d'instances de l'objet Ball
Balle[] balles = new Balle[nbreBalle];
```

Comme dans l'exemple des nombres entiers, nous venons uniquement de déclarer des copies du modèle de balle. Il faut maintenant les créer dans le `setup()`. Nous allons dessiner trois balles au centre de l'écran. Toutes les trois seront blanches.

```
void setup() {
    smooth(); //Lissage des dessins
    size(400, 200); //Taille de la fenêtre
    //DEBUT AJOUT
```

```
//Cette boucle va crée trois balles blanches au centre de
l'écran
for (int i = 0; i < nbreBalle; i++) {
  balles[i] = new Balle(width/2, height/2, color(255));
}
//FIN AJOUT
}
```

Dans la méthode `draw()`, nous allons aussi créer une boucle qui va parcourir tous les éléments de la liste pour les déplacer, tester leurs collisions et les afficher. Nous allons remplacer :

```
//Déplacement et affichage de la balle
maBalle.bouge();
maBalle.testCollision();
maBalle.display();
```

par

```
//Cette boucle va déplacer et afficher les trois balles
for (int i = 0; i < nbreBalle; i++) {
  balles[i].bouge();
  balles[i].testCollision();
  balles[i].display();
}
```

Une dernière opération va consister à modifier le constructeur du modèle de la balle afin que chaque balle ait une vitesse et une direction spécifiques. Pour

ce faire nous allons utiliser la fonction `random()` qui permet de générer des nombres aléatoires. Nous allons remplacer le constructeur ci-dessous :

```
//Constructeur de la balle
Balle (float nouvX, float nouvY, color nouvCouleur) {
  x = nouvX;
  y = nouvY;
  couleur = nouvCouleur;
  vitesseX = 2;
  vitesseY = 2;
}
```

par celui-ci :

```
//Constructeur de la balle
Balle (float nouvX, float nouvY, color nouvCouleur) {
  x = nouvX;
  y = nouvY;
  couleur = nouvCouleur;
  vitesseX = 2 + random(-1,1);
  vitessesY = 2 + random(-1,1);
}
```

Code final

Voici le programme complet :

```
//Déclaration d'une variable contenant le nombre de balles
```

```
int nbreBalle = 3;

//Déclaration d'une liste d'instances de l'objet Ball
Balle[] balles = new Balle[nbreBalle];

void setup() {
    smooth(); //Lissage des dessins
    size(400, 200); //Taille de la fenêtre

    //Cette boucle va créeé trois balles blanches au centre de
    l'écran
    for (int i = 0; i < nbreBalle; i++) {
        balles[i] = new Balle(width/2, height/2, color(255));
    }
}

void draw() {
    fill(0, 0, 0, 1);
    rect(0, 0, width, height);
    noStroke();

    //Cette boucle va déplacer et afficher les trois balles
    for (int i = 0; i < nbreBalle; i++) {
        balles[i].bouge();
        balles[i].testCollision();
        balles[i].display();
    }
}

class Balle {
    //Déclaration des caractéristiques de base de la balle
    float x;
    float y;
    float vitesseX;
    float vitesseY;
    color couleur;

    //Constructeur de la balle
    Balle (float nouvX, float nouvY, color nouvCouleur) {
        x = nouvX;
        y = nouvY;
        couleur = nouvCouleur;
        vitesseX = 2 + random(-1,1);
        vitessesY = 2 + random(-1,1);
    }

    //Dessin de la balle
    void display() {
        fill(couleur);
        ellipse(x, y, 40, 40);
    }

    void bouge() {
        x = x + vitesseX;
        y = y + vitesseY;
    }
}
```

```
void testCollision() {  
  //Si la balle touche un mur, elle rebondit  
  if (x > width-20 || x < 20) {  
    vitesseX = vitesseX *-1;  
  }  
  if (y > height-20 || y < 20) {  
    vitesseY = vitesseY * -1;  
  }  
}  
}
```

N'hésitez pas à modifier certains paramètres du programme pour vous approprier davantage son fonctionnement.

6.5 La lecture du son

Processing est un environnement principalement dédié à la création visuelle. Il n'a pas été conçu au départ pour jouer du son, et encore moins pour générer directement des ondes audio, tel un logiciel comme Pure Data. Par la suite, de nombreuses bibliothèques externes ont été créées pour étendre les capacités sonores de *Processing*. Ainsi vous pouvez à n'importe quel moment intégrer ces bibliothèques externes dans vos programmes : des synthétiseurs, des sampleurs audio, des interfaces MIDI, des séquenceurs, etc. Sachez toutefois que ce n'est pas la mission de base de *Processing* davantage centrée sur la génération de formes visuelles.

Dans ce chapitre, nous allons nous familiariser avec un *minim* de fonctionnalités audio disponibles par défaut dans *Processing*.

Minim

Il a été décidé à un moment donné d'intégrer dans toute distribution *Processing* la bibliothèque audio *Minim* pour pouvoir par défaut jouer des fichiers audio ou capter le son entrant par le microphone. A priori cette bibliothèque est déjà installée sur votre machine. Pour vérifier sa présence et l'intégrer dans votre programme, il suffit d'aller dans le menu Sketch > Import Library... > Minim audio.

Selon votre plate-forme et votre version, à la suite de cette action, *Processing* ajoutera des lignes de code plus ou moins nombreuses en haut de votre programme. Par défaut, il faut au moins voir affichée l'instruction suivante :

```
import ddf.minim.*;
```

C'est cette instruction (que vous pouvez saisir à la main si vous le voulez)

qui importera l'ensemble des fonctionnalités de la bibliothèque *Minim* pour les rendre accessibles à notre programme. C'est à partir de cette instruction `import ddf.minim.*;` que notre sketch va pouvoir jouer du son.

Configurer Minim

Le lancement de notre sketch via le bouton Run ne suffit pas pour jouer un son. Au préalable, il faut prévoir d'activer les fonctionnalités de gestion audio de *Minim* (son moteur audio en jargon informatique). Pour jouer un son, *Minim* doit en effet auparavant demander un accès à la carte son de votre ordinateur.

Ne vous inquiétez pas, *Minim* s'occupe de toute cette cuisine interne pour vous, il suffit de lui dire de s'activer pour cela. Le fait d'avoir activé *Minim* suppose de prévoir de le désactiver à la fin de notre programme. Si vous ne comprenez pas tout ce que nous venons de dire, ce n'est pas trop grave. Sachez simplement qu'il faut saisir le code suivant au début de chaque programme utilisant *Minim* :

```
import ddf.minim.*;
Minim minim;

void setup() {
    minim = new Minim(this);
}
```

Et qu'il faut également écrire la fin du programme les instructions suivantes :

```
void stop() {
    minim.stop();
    super.stop();
}
```

```
}
```

Si par curiosité vous voulez savoir à quoi sert le mot "this" dans ce programme, sachez qu'il sert à indiquer à *Processing* que nos instructions s'appliquent à "ce" programme. Minim a besoin de connaître l'adresse de "ce" programme (le nôtre) pour le faire communiquer avec la carte son de notre ordinateur. Encore une fois, si vous ne comprenez pas ce que nous venons de dire, ce n'est pas grave. Sachez seulement que ces mots doivent être écrits au début et à la fin de chaque programme lorsque nous voulons produire du son.

Importer un son

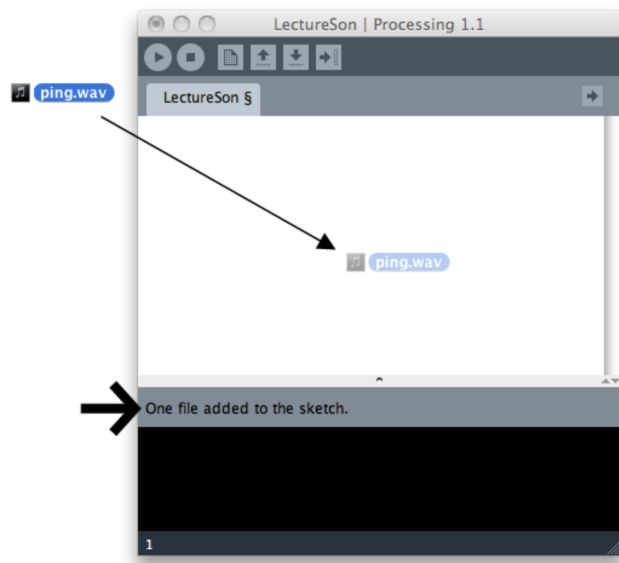


Figure 6.3: Importer un son dans *Processing*.

Tout comme les images photographiques ou la typographie, nous allons importer un fichier son dans *Processing* pour pouvoir le jouer dans notre programme. Pour bien réussir cette étape, nous vous recommandons de sauvegarder d'abord votre sketch.

Nos sons peuvent venir de plusieurs sources : des sons téléchargés, des sons que vous avez créés vous-même via un logiciel comme Audacity, voire des sons que vous pouvez trouver par défaut dans votre système d'exploitation.

Identifiez et sélectionnez à présent le fichier son que vous souhaitez utiliser, et glissez-le directement sur la fenêtre *Processing*.

Cette action placera le son de ce fichier dans le dossier "data" de votre sketch (un message s'affichant au niveau de la console de *Processing* vous le confirmera). Si vous voulez voir ce que contient ce dossier, taper sur ctrl-k (Windows/Linux) ou cmd-k (Mac), sinon choisissez dans le menu Sketch > Show Sketch Folder.

Formats audio

Vous pouvez utiliser trois formats audio avec Minim : *WAV*, *AIFF*, et *MP3*. Les deux premiers formats peuvent être exportés par défaut via des logiciels libres comme Audacity. Le troisième format nécessite à ce jour un plug-in pour Audacity, mais celui-ci est relativement facile à installer. Comme pour les images, les différents formats ont leurs avantages et désavantages.

- Les formats *WAV* et *AIFF* sont quasi identiques et sont deux formats non compressés. Ils peuvent être lus sans trop de difficultés par Minim, car ils ne requièrent pas de décompression. Ils sont souvent utilisés pour des sons courts, de type bruitage. Pour un morceau de musique de même durée, ces deux formats occupent beaucoup plus de place mémoire que les fichiers au format *MP3*.

- Le format *MP3* est un format compressé. Il est souvent beaucoup moins volumineux que les fichiers *WAV* et *AIFF*. Par contre, la lecture du format *MP3* nécessite plus de ressources de calcul de votre ordinateur, car le fichier doit être décompressé lors de sa lecture.

Une horloge

Voici un programme simple, qui anime une ellipse avec les secondes de l'horloge de votre ordinateur (l'horloge système). Nous allons utiliser ce programme comme base pour créer une horloge qui sonne toutes les minutes. Pour mieux anticiper cet événement qui se déroule toutes les soixante secondes - car il nécessite une certaine patience - nous allons représenter les secondes par une ellipse qui s'agrandit progressivement depuis le centre de l'espace de dessin.

```
void draw() {
  background(255);
  ellipse(50,50,second(),second());
}
```

Nous utiliserons la valeur des secondes de l'horloge système pour définir les paramètres largeur, hauteur de l'ellipse. Si les secondes sont égales à 42, nous aurons une ellipse de 42 x 42 pixels. Si les secondes sont égales à 2, nous aurons une ellipse de 2 x 2 pixels. Nous pouvons également ajouter une condition pour signaler davantage le changement des minutes : à chaque fois que les secondes reviendront à zéro, nous marquerons l'événement avec un écran rouge.

```
void draw() {
```

```
  if (second() == 0) {
    background(255,0,0);
  } else {
    background(255);
  }
  ellipse(50,50,second(),second());
}
```

La condition `if(second() == 0)` compare les secondes avec la valeur 0. Si les deux sont égales, le fond sera rouge, sinon (`else`) le fond sera blanc.

Une horloge sonore

Nous allons maintenant sonoriser ce programme en jouant un son à chaque fois que les secondes de notre horloge reviennent à zéro. Pour commencer, nous allons ajouter autour de notre méthode `draw()` de l'horloge, toutes les méthodes nécessaires pour la sonoriser.

```
import ddf.minim.*; //1)
Minim minim; //2)
AudioSnippet ping;

void setup() {
  minim = new Minim(this); //3)
  ping = minim.loadSnippet("ping.wav"); //4)
}

void draw() {
  if (second() == 0) {
```

```

    if (ping.isPlaying() == false){ //5)
        ping.play(); }
    background(255,0,0);
  } else {
    background(255);
  }
  ellipse(50,50,second(),second()); //6)
}

void stop() {
  ping.close(); //7)
  minim.stop();
  super.stop(); //8)
}

```

Attardons-nous sur la structuration de ce sketch : ce programme nécessite de nombreuses étapes pour fonctionner :

- Importer les fonctionnalités Minim dans notre sketch.
- Créer deux variables dont l'une qui contiendra le moteur Minim et l'autre les données audio.
- Démarrer le moteur Minim au début du programme.
- Copier les données du fichier audio dans notre variable.
- Actions conditionnelles : jouer le son et afficher un écran rouge lorsque les secondes sont égales à zéro (vérifier que ce son n'est pas déjà en train de jouer) .
- Dessiner une ellipse en fonction de la valeur des secondes de l'horloge.
- A la fin du programme, arrêter le son.
- A la fin du programme, arrêter Minim.

Nous avons déjà expliqué le démarrage et l'extinction du moteur Minim. Nous n'y revenons pas. Par contre, nous attirons votre attention sur les parties 2, 4 et 5 du programme qui concernent l'importation et la lecture du son.

Tout d'abord, nous avons créé une variable de type `AudioSnippet`. Le type `AudioSnippet` est un type de variable défini à l'intérieur du code de Minim. Nous vous rappelons que dans le cas de n'importe quelle variable, la formule pour la déclarer est de la forme `type de la variable nom de la variable = les valeurs de la variable`. Par exemple, si par un tour de force il était possible d'importer un petit oiseau tout mignon dans *Processing*, il suffirait d'écrire `PetitOiseau pioupiou = loadBird("pioupiou.bird");`. En d'autres termes, on écrit d'abord le type de la chose, le nom de la chose, et enfin on lui donne sa valeur. Pour revenir à notre exemple d'horloge sonore, cette valeur est donnée par la fonction `loadSnippet()` qui va aller chercher les valeurs d'ondes dans le fichier audio et les importera dans notre variable nommée "ping".

Pour plus d'informations sur les variables et les différents types de variables, reportez-vous au chapitre dédié à ce sujet. Une fois la variable chargée avec le son, il suffit de le jouer. Dans le cas du *AudioSnippet* la commande pour jouer un son s'appelle `play()`. Enfin, il peut être important de vérifier à un moment donné si le son n'est pas déjà en train d'être joué. C'est le rôle de la commande `isPlaying()` de nous dire si oui (`true`) ou non (`false`) nous sommes dans cette situation. Dans notre exemple, si nous avons oublié cette condition, le son se serait lancé pendant soixante fois durant la totalité de la seconde où `second()` est égal à zéro.

AudioSnippet, AudioSample, AudioPlayer

La bibliothèque Minim distingue plusieurs manières de jouer un son dans *Processing*. En principe, les instructions à utiliser dépendront de la nature et rôle du son dans votre programme. Si l'on se réfère à la documentation dé-

crivant les fonctionnalités offertes par Minim, un AudioSnippet sera employé pour une lecture simple d'un court extrait sonore, un AudioSample pour une lecture rapide et répétée comme une boîte à rythmes, et enfin un AudioPlayer pour jouer des fichiers audio plus longs (souvent de type MP3) situés directement sur le disque dur de votre ordinateur pour ne pas alourdir encore davantage la mémoire interne du programme. Voici un exemple qui utilise les trois méthodes de lecture du son associées respectivement à trois fichiers audio imaginaires dénommés ping, pop et song.

```
import ddf.minim.*;
Minim minim;
AudioSample ping;
AudioSnippet pop;
AudioPlayer song;

void setup() {
    minim = new Minim(this);
    ping = minim.loadSample("ping.aiff");
    pop = minim.loadSnippet("pop.aiff");
    song = minim.loadFile("song.mp3");
}

void draw() {
}

void keyPressed() {
    if (key == 'a') ping.trigger();
    if (key == 'b') {
        pop.rewind();
```

```
        pop.play();
    }
    if (key == 'c') song.play();
}

void stop() {
    ping.close();
    pop.close();
    song.close();
    minim.stop();
    super.stop();
}
```

Notez avant tout que les instructions d'importation et de lectures changent selon le type choisi. AudioPlayer et AudioSample utiliseront tous les deux la méthode play() car leur lecture est toujours unique (un seul son à la fois). Par contre, AudioSample nécessite la lecture de nombreux sons simultanés (on peut taper plusieurs fois sur une batterie) et pour cette raison utilise une autre nomenclature, trigger(), pour décrire le lancement d'un son. En anglais, le mot "trigger" signifie une gâchette.

Documentation

Pour plus d'informations sur les différentes fonctionnalités de Minim, ainsi qu'un grand nombre d'exemples d'utilisation (avec leur code source librement mis à disposition), reportez-vous au site <http://code.compartmental.net/tools/minim/> ou à la documentation technique (plus difficile à lire) consultable à l'adresse <http://code.compartmental.net/minim/javadoc/ddf/minim/package-tree.html>.

Interagir avec *Processing*

Sommaire

7.1 Les évènements clavier	99
7.2 Les événements souris	102
7.3 L'entrée microphone	106
7.4 L'entrée vidéo	110

Capter les actions de l'utilisateur et s'en servir pour faire réagir votre programme constitue une première forme d'interactivité.

7.1 Les évènements clavier

Les actions de l'utilisateur sur le clavier de votre ordinateur sont appelés les évènements clavier.

Nous allons découvrir dans ce chapitre comment récupérer les informations liées au clavier en fabriquant une machine à écrire très simple qui nous permettra d'imprimer du texte sur l'écran.

Actions

La valeur de la dernière touche appuyée est stockée dans la variable `key` proposée par défaut dans *Processing*. Cette variable ne peut stocker qu'un seul caractère à la fois, symbolisé par l'écriture '`a`', '`b`', '`c`', ... Attention, `key` est sensible à la casse, c'est-à-dire qu'il fait une différence entre les minuscules et majuscules. Dans l'exemple suivant, nous allons afficher dans la fenêtre de visualisation le caractère correspondant à la touche qui aura été appuyée sur le clavier de votre ordinateur. Pour ce faire, nous employons la méthode `text()`.

```
void draw() {
  background(204);
```

```
fill(0);
textSize(70);
text(key,30,70);
}
```

Par ailleurs, nous pouvons capter dans notre programme le moment où l'utilisateur a appuyé ou relâché une touche du clavier, par le biais des méthodes `keyPressed()` et `keyReleased()`. Ces deux méthodes vont être automatiquement appelées par *Processing* au moment où l'état d'une touche change. Dans l'exemple suivant, `keyPressed()` et `keyReleased()` changent la valeur de la variable `y`, utilisée pour positionner le caractère à l'écran selon que la touche est appuyée ou relâchée.

```
int y = 0;

void setup() {
  size(130,200);
  textSize(80);
  stroke(0);
  fill(0);
}

void draw() {
  background(204);
  line(10,100,120,100);
  text(key,35,y);
}
```

```
void keyPressed() {
  y = 180;
}
void keyReleased() {
  y = 80;
}
```

Touches spéciales

Les touches spéciales comme les flèches (UP, DOWN, LEFT, RIGHT) ou ALT, CONTROL, SHIFT sont quant à elles stockées dans la variable `keyCode`. Le test `if (key == CODED)` permet de vérifier si la touche appuyée est une touche spéciale ou non. Dans un programme, il faudra distinguer les deux cas en fonction du type de touche appuyée que l'on veut tester. Dans l'exemple suivant, nous allons créer une machine à écrire un peu particulière, puisqu'il sera possible de déplacer le texte grâce aux flèches. A chaque fois qu'une touche sera appuyée, elle sera stockée dans une chaîne de caractère, pour pouvoir être affichée dans le `draw()` à l'aide de la méthode `text()`.

```
String s = "";
int x = 50;
int y = 50;

void setup() {
  size(200,200);
}

void draw() {
```

```
background(255);
fill(0);
text(s, x, y);
}

void keyPressed() {
  if (key == CODED) {
    if(keyCode==LEFT) x =x-1;
    if(keyCode==RIGHT) x=x+1;
    if (keyCode == UP) y = y - 1;
    if(keyCode==DOWN) y=y+1;
  } else {
    s = s + key;
  }
}
```

un carré si la touche appuyée est 'r'.

```
void draw() {
  rectMode(CENTER);
  background(128);
  if (keyPressed == true) {
    if (key == 'c') {
      ellipse(50,50,90,90);
    } else if (key == 'r') {
      rect(50,50,90,90);
    }
  }
}
```

Notons l'omission des accolades après chaque `if`. Cette syntaxe est possible si le `if` ne s'applique qu'à une seule instruction. Nous utilisons par ailleurs une variable de type `String` dénommé dans cet exemple "s" dont la fonction est de stocker une suite de caractères, comme des phrases, par exemple.

Vérification spécifique de l'appui d'une touche

Processing nous permet également de faire appel à la variable `keyPressed` et de la définir. Attention ! Bien que la syntaxe soit identique, elle n'est pas à confondre avec la méthode `keyPressed()` abordée au début de ce chapitre. Cette variable va nous informer en permanence si une touche est appuyée ou non et pourra être utilisée dans la méthode `draw()` notamment. Dans l'exemple suivant, nous allons dessiner un cercle si la touche appuyée est 'c',

7.2 Les événements souris

Dans ce chapitre, nous allons voir comment interagir avec la souris, en récupérant des informations comme sa position ou bien les actions de clics. Pour illustrer ces fonctionnalités, nous allons créer un petit logiciel de dessin, à base de cercles, en introduisant même du mouvement aléatoire !

Position de la souris

Les coordonnées de la souris dans la fenêtre sont accessibles par les deux variables `mouseX` et `mouseY`, disponibles par défaut dans *Processing*. Elles permettent de connaître la position de la souris par rapport à notre fenêtre de dessin, en prenant pour origine le coin gauche supérieur de la fenêtre. Dans l'exemple suivant, nous allons créer un programme qui va dessiner un cercle à partir de la position de la souris.

```
void setup() {  
    size(200,200);  
    smooth();  
}  
  
void draw() {  
    ellipse(mouseX,mouseY,15,15);  
}
```

A présent, nous allons modifier très légèrement l'exemple précédent en changeant dynamiquement le rayon du cercle. Celui-ci est choisi au hasard grâce à la méthode `random()`, qui génère un nombre aléatoirement au sein d'une plage de valeur. A chaque fois que *Processing* va exécuter `draw()`, la variable

`r` sera remplie avec un nombre au hasard compris entre 3 et 30 : 12, 25, 23, 11, 22, 4, 10, 11, 25 ... Le changement de valeur du rayon va provoquer le tremblement du cercle autour du curseur de la souris.

```
void setup() {  
    size(300,300);  
    smooth();  
}  
  
void draw() {  
    float r = random(3,30);  
    ellipse(mouseX,mouseY,r,r);  
}
```

Dès que la souris sort de la fenêtre, la position de la souris n'est plus relayée à notre programme. Les cercles se dessinent et s'accumulent à la dernière position captée par notre sketch.

Clics de souris

Nous pouvons intercepter les clics de souris grâce aux méthodes `mousePressed()` et `mouseReleased()`. Ces deux méthodes permettent de savoir si l'utilisateur a appuyé ou relâché un des boutons de la souris. Reprenons l'exemple précédent en changeant la couleur de remplissage de notre cercle lorsque l'utilisateur appuie sur un des boutons de la souris. Nous allons choisir un niveau de gris pris au hasard au moment du clic que nous appliquerons au cercle à l'aide de la commande `fill()`.

```
float ton_de_gris = 255;
```

```
void setup() {  
  size(300, 300);  
  smooth();  
}  
  
void draw() {  
  float r = random(10, 80);  
  fill(grey);  
  ellipse(mouseX, mouseY, r, r);  
}  
  
void mousePressed() {  
  grey = random(255);  
}
```

Processing nous permet d'identifier quel bouton de la souris a été appuyé. Pour cela, nous pouvons utiliser la variable `mouseButton` qui va contenir soit `LEFT`, `RIGHT` ou `CENTER` correspondant au bouton droit, gauche et central (si la souris dispose de tous ces boutons). Utilisons cette variable dans un exemple pour dessiner un cercle qui change de couleur en fonction du type de bouton pressé.

```
void setup() {  
  size(200,200);  
  smooth();  
}  
  
void draw() {
```

```
  background(255);  
  ellipse(100,100,200,200);  
}  
  
void mousePressed() {  
  if (mouseButton == LEFT) fill(0);  
  if (mouseButton == RIGHT) fill(255);  
  if (mouseButton == CENTER) fill(128);  
}
```

Autres événements

Processing permet de capter deux autres événements souris, notamment lorsqu'elle est en mouvement au-dessus de la fenêtre. L'instruction `mouseMoved()` va permettre de détecter le mouvement de la souris lorsque celle-ci se déplace au-dessus de la fenêtre de dessin. Si le pointeur de souris sort de la zone de la fenêtre ou s'il ne bouge plus, alors la méthode n'est plus appelée. `mouseDragged()` est appelée lorsque l'utilisateur a cliqué sur un bouton tout en bougeant la souris au-dessus de la fenêtre. Cette instruction permet notamment de gérer le glisser-déposer, par exemple. L'instruction `mouseDragged()` continue d'être actif même si la souris sort de la fenêtre. Dans l'exemple suivant, nous allons utiliser ces méthodes. Lorsque `mouseMoved()` est appelé suite à l'action de l'utilisateur, on change la couleur de remplissage du cercle et quand c'est `mouseDragged()` qui l'est, on ajuste sa taille.

```
fill(0);  
int r = 100;  
int c = 100;
```

```

void setup() {
  size(255, 255);
  smooth();
}

void draw() {
  background(255);
  fill(c);
  ellipse(width/2, height/2, r, r);
}

void mouseMoved() {
  c = mouseY;
}

void mouseDragged() {
  r = mouseX;
}

```

Curseur

Parfois il est bien pratique de cacher le curseur de la souris, par exemple dans le cas d'une installation artistique où le pointeur pourrait venir malencontreusement parasiter l'affichage de votre animation. Dans *Processing*, il existe fort heureusement une instruction pour masquer le pointeur :

```
noCursor();
```

Il est également possible de modifier la forme du curseur pour signaler certains événements à l'utilisateur, par exemple changer la forme du curseur au survol d'un élément. Il suffit pour cela d'utiliser `cursor()` avec comme paramètre une des valeurs suivantes : `ARROW`, `CROSS`, `HAND`, `MOVE`, `TEXT`, `WAIT`. L'exemple ci-après affiche une forme différente de curseur (une flèche, une croix, une main, un sablier) selon la zone de dessin survolée par la souris.

```

void setup() {
  fill(255);
  rect(10, 10, 20, 80);
  fill(175);
  rect(40, 10, 20, 80);
  fill(100);
  rect(70, 10, 20, 80);
}

void draw() {
  // Nous vérifions si on survole un des 3 rectangles
  // et modifions le curseur en conséquence

  if (mouseX>10 && mouseX<30 && mouseY>10 && mouseY<90)
    cursor(HAND); // Affiche une main
  } else if (mouseX>40 && mouseX<60 && mouseY>10 && mouseY<90)
    cursor(ARROW); // Affiche une flèche
  } else if (mouseX>70 && mouseX<90 && mouseY>10 && mouseY<90)
    cursor(WAIT); // Affiche un sablier
  } else {
    cursor(CROSS); // Affiche une croix si on ne survole rien
  }
}

```

```
}  
}
```

7.3 L'entrée microphone

Si votre sketch peut être programmé pour être sensible à l'appui d'une touche du clavier ou au clic de la souris, ce n'est pas la seule forme d'interactivité offerte par *Processing*. Vous pouvez également donner à votre programme le sens de l'ouïe en vous servant d'un microphone comme oreille et en faisant correspondre des actions du sketch à certains paramètres du son capté. Nous allons tout d'abord rappeler certaines notions abordées dans le chapitre consacré à la lecture du son pour ensuite nous concentrer plus spécifiquement sur l'ajout d'une source sonore à l'aide d'un microphone.

Minim

Il a été décidé à un moment donné d'intégrer dans toute distribution *Processing* la bibliothèque audio *Minim* pour pouvoir jouer au moins des fichiers audio ou capter le son entrant par le microphone. A priori cette bibliothèque (également appelé librairie en jargon informatique) est déjà installée sur votre ordinateur. Pour vérifier sa présence et l'intégrer dans votre programme, il suffit d'aller dans le menu *Sketch > Import Library... > Minim audio*. Selon votre plate-forme et votre version, à la suite de cette action, *Processing* ajoutera des lignes de code plus ou moins nombreuses en haut de votre programme. Par défaut, il faut au moins voir affichée l'instruction suivante :

```
import ddf.minim.*;
```

C'est cette instruction (que vous pouvez saisir à la main si vous le voulez) qui importera l'ensemble des fonctionnalités de la bibliothèque *Minim* pour les rendre accessibles à notre programme. C'est à partir de cette instruction `import ddf.minim.*;` que notre sketch va pouvoir jouer du son.

Configurer *Minim*

Le lancement de notre sketch à l'aide du bouton **Run** ne suffit pas pour jouer un son. Au préalable, il faut prévoir d'activer les fonctionnalités de gestion audio de *Minim* (son moteur audio en jargon informatique). Pour jouer un son, *Minim* doit en effet auparavant demander un accès à la carte son de votre ordinateur. Ne vous inquiétez pas, *Minim* s'occupe de toute cette cuisine interne pour vous, il suffit de lui dire de s'activer pour cela. Le fait d'avoir activé *Minim* suppose de prévoir de le désactiver à la fin de notre programme.

Si vous ne comprenez pas tout ce que nous venons de dire, ce n'est pas trop grave. Sachez simplement qu'il faut saisir le code suivant au début de chaque programme utilisant *Minim* :

```
import ddf.minim.*;
Minim minim;

void setup() {
    minim = new Minim(this);
}
```

..et qu'il faut également écrire à la fin du programme les instructions suivantes :

```
void stop() {
    minim.stop();
    super.stop();
}
```

Ajouter une source sonore

Pour exploiter l'entrée microphone, nous devons créer un objet *source sonore* qui permettra d'accéder au son capté par le microphone.

```
import ddf.minim.*;
Minim minim;
AudioInput in;

void setup() {
  minim = new Minim(this);
  in = minim.getLineIn(Minim.STEREO, 512);
}
```

Il faut également prévoir d'arrêter la capture du son ainsi que la librairie *Minim* à la fin de l'exécution du programme. A cet effet, nous employons la méthode `stop()` :

```
void stop() {
  in.close();
  minim.stop();
  super.stop();
}
```

Visualiser le niveau sonore

L'objectif à présent est de faire varier la couleur du fond de la fenêtre de l'espace de dessin en fonction des sons captés par le micro. Plus le son sera

fort, plus le fond sera blanc. Nous allons utiliser l'objet source audio (`in`), qui comporte trois sous objets : `left`, `right` et `mix`. Dans le cas d'un microphone stéréo, ces trois éléments représentent le canal gauche, droit et la moyenne des deux. A partir de ces canaux, il est possible de connaître le niveau sonore du son capté à l'aide de la méthode `level()` qui retourne au programme cette information sous la forme d'une valeur chiffrée.

```
void draw() {
  background(in.mix.level()*2550);
}
```

Un petit jeu

Nous allons à présent créer un jeu très simple qui va exploiter davantage les possibilités d'interaction avec un microphone. Le jeu sera composé d'une balle partant de la gauche de l'écran et qui subit une force d'attraction lente, mais permanente vers cette partie de l'espace de dessin. En faisant du bruit, l'utilisateur va la pousser vers la droite, son but étant de passer la ligne d'arrivée. Pour commencer, nous allons changer la taille de notre fenêtre de visualisation de l'espace de dessin, activer le lissage du tracé et définir le contour des formes en lui appliquant la couleur blanche. Le code concerné est signalé en gras.

```
void setup() {
  size(600, 100);
  smooth();
  stroke(255);
```

```

    minim = new Minim(this);
    in = minim.getLineIn(Minim.STEREO, 512);
}

```

Nous allons ensuite créer une variable qui stockera la position de la balle. Elle sera déclarée en en-tête du programme pour être disponible dans l'ensemble du sketch. Nous lui assignons la valeur 0.

```

float ballX = 0;
void setup() {
...

```

Dans la méthode `draw()`, nous allons définir un fond noir et dessiner une balle qui réagit en fonction du niveau du son. A chaque appel de la méthode `draw()` nous ajoutons le niveau sonore du microphone à la coordonnée x de la balle. Elle va se mettre en mouvement au fur et à mesure que nous faisons du bruit.

```

void draw() {
    background(0);
    ballX = ballX + in.mix.level()*20;
    ellipse(25 + ballX, height - 25, 50, 50);
}

```

Pour éviter que la balle ne sorte de l'écran, nous allons ajouter deux conditions qui vont corriger sa position si la variable `ballX` est plus petite que 0 ou plus grande que la largeur du sketch.

```

void draw() {

```

```

    background(0);
    ballX = ballX + in.mix.level()*20;
    if (ballX < 0) {
        ballX = 0;
    }
    if (ballX > width-25) {
        ballX = width-25;
    }

    ellipse(25+ballX, height-25, 50, 50);
}

```

Nous allons ajouter une ☐ ligne d'arrivée ☐ ainsi qu'une condition qui spécifie que si la balle l'a franchi, elle change de couleur.

```

void draw() {
    background(0);

    ballX = ballX + in.mix.level()*20;
    if (ballX < 0) {
        ballX = 0;
    }
    if (ballX > width-25) {
        ballX = width-25;
    }

    if (ballX > 500) {
        fill(255, 0, 0);

```

```
} else {  
    fill(255);  
}  
line(500, 0, 500, 100);  
ellipse(25+ballX, height-25, 50, 50);  
}
```

Afin de compliquer le jeu, nous allons ajouter un comportement à la balle qui la fait revenir en arrière en permanence. A chaque appel de la fonction draw() nous allons diminuer légèrement sa position sur l'axe x.

```
void draw() {  
    background(0);  
  
    ballX = ballX - 0.5;  
    ballX = ballX + in.mix.level()*20;  
    if (ballX < 0) {  
        ballX = 0;  
    }  
    if (ballX > width - 25) {  
        ballX = width - 25;  
    }  
    if (ballX > 500) {  
        fill(255, 0, 0);  
    } else {  
        fill(255);  
    }  
    line(500, 0, 500, 100);
```

```
        ellipse(25 + ballX, height - 25, 50, 50);  
    }
```

A présent, il ne vous reste plus qu'à faire appel à votre imagination pour utiliser du son dans vos projets en exploitant les nombreuses possibilités : analyse des fréquences d'un son, création d'effets sonores sur mesure à l'aide de fonctionnalités de synthèse audio, etc.

7.4 L'entrée vidéo

Processing permet de capturer les images provenant d'une caméra vidéo connectée à votre ordinateur par un câble ou à distance via le Wifi ou même Internet. Ces images peuvent ensuite être affichées dans la fenêtre de visualisation de votre sketch et le cas échéant, modifiées en fonction de vos besoins. Les applications créatives de l'entrée vidéo sont multiples.

Les types de caméras supportées

Les caméras les plus simples à utiliser sont les caméras USB, souvent appelées caméras Web. Il est également possible d'employer des caméras que l'on branche sur un port IEEE1394 (Firewire) comme les caméras DC1394 et les caméras DV. Pour exploiter l'entrée vidéo, vous pouvez utiliser la librairie *video* de *Processing* qui fait appel à la librairie QuickTime, de la compagnie Apple Computer. Toutes les caméras supportées par QuickTime devraient donc fonctionner sans aucune difficulté. Apple fournit cette librairie uniquement pour les systèmes d'exploitation Mac OS X et Microsoft Windows. Sur GNU/Linux, il est préférable d'utiliser la librairie GSVideo. Celle-ci fait usage de l'excellente GStreamer.

Capturer des images avec la librairie vidéo

Sous réserve que vous soyez sur Mac OS X ou Windows, voyons à présent comment utiliser la librairie *video*. Tout d'abord il faut importer la librairie vidéo en cliquant dans le menu : *Sketch > Import Library >... > Video*. Une ligne de code s'affichera au début de votre programme :

```
import Processing .video.*;
```

Ensuite nous allons déclarer une variable qui stockera notre objet Camera. Dans la méthode `setup()`, il faut préciser laquelle des caméras présentes sera utilisée. Notez que vous devez déjà avoir connecté et installer cette caméra vidéo sur votre ordinateur pour faire fonctionner cet extrait de code. Si le code génère une erreur au moment de créer la caméra, essayez d'autres numéros de devices[].

```
Capture camera;
```

```
void setup() {
    size(640, 480);
    background(0);

    String[] devices = Capture.list();
    println(devices);
    camera = new Capture(this, 320, 240, devices[0]);
}
```

L'étape suivante consiste à préciser, dans la méthode `draw()`, que nous affichons la dernière image obtenue par la caméra en vérifiant qu'une nouvelle est bien disponible. En effet, les images sont reçues de la caméra vidéo à une certaine fréquence, par exemple 30 fois par seconde, ce qui ne correspond pas forcément à celle de notre sketch. Dans l'exemple ci-après, nous avons demandé à notre programme d'appliquer sur l'image captée un filtre de solarsation (inversion des valeurs d'ombre et de lumière de l'image).

```
void draw() {
```

```
if (camera.available()) {  
  camera.read();  
  camera.filter(INVERT);  
  image(camera, 160, 100);  
}  
}
```

La variable de type `Capture` peut être utilisée de la même manière qu'une variable de type `PImage` : il est possible de lire les pixels de l'image, la transformer, l'afficher plusieurs fois à l'écran, et bien d'autres choses encore. Le programme complet de notre exemple :

```
import Processing .video.*;  
Capture camera;  
  
void setup() {
```

```
  size(640, 480);  
  background(0);  
  String[] devices = Capture.list();  
  println(devices);  
  camera = new Capture(this, 320, 240, devices[0]);  
}
```

```
void draw() {  
  if (camera.available()) {  
    camera.read();  
  
    camera.filter(INVERT);  
    image(camera, 160, 100);  
  }  
}
```


Exposer ses projets *Processing*

Sommaire

8.1 L'exportation	113
8.2 L'impression	115
8.3 La vidéo	118

Pour l'instant, nous avons travaillé exclusivement dans l'environnement de Processing. Nous tapons des lignes de code et nous les exécutons depuis l'interface par le bouton Run. A présent, nous allons voir que Processing intègre deux manières d'exporter notre programme pour le distribuer. L'une, à destination du web, permet d'exposer son animation en ligne, visible sur une page dans un navigateur. L'autre procédé de diffusion permet de créer une application autonome fonctionnant directement sur un ordinateur utilisant Windows, Mac ou Linux. Dans les deux cas, plus besoin de l'éditeur pour faire tourner votre sketch.

8.1 L'exportation

Pour l'instant, nous avons travaillé exclusivement dans l'environnement de Processing. Nous tapons des lignes de code et nous les exécutons depuis l'interface par le bouton **Run**. A présent, nous allons voir que *Processing* intègre deux manières d'exporter notre programme pour le distribuer. L'une, à destination du web, permet d'exposer son animation en ligne, visible sur une page dans un navigateur. L'autre procédé de diffusion permet de créer une application autonome fonctionnant directement sur un ordinateur utilisant Windows, Mac ou Linux. Dans les deux cas, plus besoin de l'éditeur pour faire tourner votre sketch.

Exporter pour le web

Une fois que votre programme fonctionne et que vous êtes satisfait du résultat, vous pouvez générer un *applet*, objet intégrable à une page web et lisible par votre navigateur internet habituel. Pour effectuer applet, objet intégrable à une page web et lisible par votre navigateur internet habituel. Pour effectuer cette exportation, il suffit de cliquer sur le bouton **Export**.

L'opération effectuée, *Processing* ouvre automatiquement un dossier *applet* dans l'explorateur de fichiers de votre ordinateur. Celui-ci contient plusieurs fichiers.

- index.html est la page web générée par *Processing*. En double cliquant dessus, votre navigateur internet va se lancer et afficher l'animation que vous venez juste d'exporter.
- loading.gif est une icône utilisée lors du chargement de l'applet dans la page web.
- *votre nom de sketch.jar* est l'*applet* à proprement parler. Il s'agit du code source de votre sketch qui a été transformé et sauvegardé avec toutes les données.
- *votre nom de sketch.java* est la version Java de votre code.
- *votre nom de sketch.pde* est le code source original du sketch que vous avez saisi dans l'éditeur de *Processing*.

Attention ! A chaque exportation, *Processing* écrase automatiquement les fichiers portant le même nom. Par précaution, si vous effectuez des modifications sur le fichier *votre nom de sktech.html*, pensez au préalable à le renommer.

Si vous disposez d'un espace de stockage sur un serveur web, et que vous êtes familiers des transferts FTP, vous pouvez téléverser le dossier *applet* (en le renommant, si besoin est) dans votre espace en ligne. Vous pourrez ainsi rendre accessibles vos sketches sur Internet.

Exporter pour l'ordinateur

Votre sketch *Processing* peut également être exporté sous la forme d'une application directement exécutable sur Windows, Mac et GNU/Linux. Il suffira alors de cliquer dessus pour le lancer comme n'importe quel autre logiciel présent sur votre ordinateur. Pour effectuer cette exportation, il suffit de sélectionner dans le menu **File > Export Application**. La fenêtre *Export Options* devrait s'ouvrir.

- *Platforms* permet de sélectionner la plateforme. C'est là toute la beauté de *Processing* : nous pouvons exporter une application développée sur Mac pour GNU/Linux, par exemple.
- *Full Screen (Present mode)* lancera l'application en plein écran lors du double-clic.
- *Show a Stop button*, en mode *Full Screen*, affiche un bouton **Stop** sur l'écran pour arrêter l'application.

Une fois l'application assemblée, la fenêtre du dossier du sketch s'affiche avec les dossiers propres à chaque plateforme. Le fichier exécutable se trouve à l'intérieur, prêt à être lancé sur votre ordinateur. Comme vous pouvez le constater, les procédures d'exportation sont vraiment très simples à réaliser.

8.2 L'impression

Pour l'instant, nous avons travaillé sur l'écran en générant des images et des animations. Nous allons maintenant nous intéresser aux possibilités qu'offre *Processing* en matière d'impression : nous allons créer un document PDF (*Portable Document Format*) contenant des formes géométriques vectorielles, document qui pourra ensuite être imprimé sur du papier ou un autre médium.

Mode direct

Dans ce premier exemple, nous allons générer 500 cercles aléatoirement et les dessiner dans un document PDF, qui sera sauvé dans notre dossier de travail.

```
import processing.pdf.*;
size(400, 400, PDF, "Cercles.pdf");
for (int i = 0; i < 500; i++) ellipse(random(width),random(height),
50, 50);
exit();
```

Comment fonctionne ce programme ? Tout d'abord, nous commençons par importer toutes les classes de la librairie *processing.pdf*. Nous utilisons la méthode *size()* qui permet de définir les caractéristiques de notre espace de dessin : cette fois-ci, nous mentionnons quatre paramètres. Notez le mot-clé *PDF* qui indique à *Processing* que nous allons dessiner dans le document "Cercles.pdf", passé en quatrième paramètre de la méthode. Avec ce mode direct de tracé, puisque nous dessinons directement dans le document et non plus sur l'écran, nous n'avons plus besoin de la fenêtre de visualisation de

Processing. Le programme se termine par la commande *exit()*, qui indique au logiciel de terminer le programme tout seul, exactement comme si nous avions appuyé sur le bouton stop de l'interface. Le document généré au format pdf se trouve dans le dossier de sketch de votre application. Vous pouvez double-cliquer dessus pour l'ouvrir.

Mode enregistrement

Nous allons voir maintenant comment dessiner à la fois sur l'écran et dans un fichier .pdf à l'aide des commandes *beginRecord()* et *endRecord()*. Ces deux méthodes vont nous permettre de démarrer et arrêter l'enregistrement des commandes graphiques dans un fichier PDF. Comme précédemment, le fichier généré est sauvegardé dans le dossier du sketch. Dans l'exemple suivant, nous allons réaliser un outil graphique, une sorte de pinceau numérique dont la taille du tracé dépend de la vitesse à laquelle nous déplaçons la souris. Au démarrage de l'application, dans le *setup()*, nous indiquons le début de l'enregistrement par la commande *beginRecord()* qui prend en paramètres le type d'export (pour l'instant *Processing* ne supporte que l'extension PDF) et le nom du fichier en second paramètre. Lorsqu'on appuie sur la touche ESPACE, l'enregistrement est stoppé et l'application s'arrête. Comme précédemment, le fichier généré est sauvegardé dans le dossier du sketch.

```
import processing.pdf.*;
boolean drag = false;

void setup() {
    size(400, 400);
    smooth();
    beginRecord(PDF, "Cercles.pdf");
```

```

}

void draw() {
  if (drag) {
    float r = dist(pmouseX, pmouseY, mouseX, mouseY) + 5;
    ellipse(mouseX, mouseY, r, r);
  }
}

void mouseDragged() {
  drag = true;
}

void mouseReleased() {
  drag = false;
}

void keyPressed() {
  endRecord();
  exit();
}

```

La ligne suivante : `float r = dist(pmouseX, pmouseY, mouseX, mouseY) + 5;` permet de calculer la vitesse de la souris grâce à la méthode `dist` et aux variables `pmouseX`, `pmouseY` et `mouseX` et `mouseY`. Nous ajoutons 5 pour ne pas avoir des cercles de rayon 0 sur l'écran (ce qui reviendrait à avoir des cercles invisibles). La variable `drag` sert à savoir si l'utilisateur est en train de déplacer la souris en maintenant le clic appuyé, ce qui active à l'aide de `draw()` l'action

de dessiner des cercles sous le pointeur de la souris.

Un générateur de livres

Au lieu de générer un document d'une seule page, nous allons à présent réaliser un livre de plusieurs pages. Nous allons créer dynamiquement un folioscope (<http://fr.wikipedia.org/wiki/Folioscope>) dont chaque page représentera une image de l'animation à un instant donné. C'est la méthode `nextPage()` qui va créer une page vierge à chaque fois que cette instruction sera appelée durant l'enregistrement.

```

import processing.pdf.*;
PGraphicsPDF pdf;
float x = 0;
float y = 75;

void setup() {
  size(400, 150);
  smooth();
  pdf = (PGraphicsPDF) beginRecord(PDF, "Bulle.pdf");
}

void draw() {
  background(255);
  fill(0);
  ellipse(x, y, 30, 30);

  x = x + random(5);
  y = y + random(-5, 5);
}

```

```
if (x - 30 > width) {  
  endRecord(); exit();  
}  
pdf.nextPage();  
}
```

Nous avons introduit la variable `pdf` pour stocker une référence au document que nous générons. Lorsque nous avons fini de dessiner une image dans `draw()`, nous passons à l'image suivante par la commande `pdf.nextPage()`. Le programme est conçu pour s'arrêter lorsque la bulle sort de l'écran. Après l'exécution du programme, le fichier sauvegardé du folioscope se trouve dans le dossier du sketch et porte le nom `Bulle.pdf`. En l'ouvrant, vous pourrez constater qu'il comporte 164 pages, chacune d'entre elles présentant notre balle en un instant donné de son mouvement dans l'espace de dessin.

Pixel versus vectoriel

Lorsque nous dessinons sur l'écran, un rectangle par exemple, nous remplissons les pixels d'une zone de l'écran avec une couleur particulière. Lorsque nous sauvegardons cette image dans un fichier, la première méthode consiste à utiliser des formats d'image particuliers (JPEG, GIF, TIFF par exemple) qui enregistrent individuellement chaque pixel qui compose l'illustration. Avec ce type de format, si nous voulons agrandir l'image, l'ordinateur sera dans l'impossibilité de créer des pixels intermédiaires et d'ajouter du détail, car il ne possède aucune information dans le fichier lui permettant d'extrapoler les modifications liées à cet agrandissement de formes.

En enregistrant la même image au format vectoriel, *Processing* ne va plus sauver chaque pixel un à un, mais plutôt les caractéristiques de chaque forme

dessinée : sa forme, sa position et sa couleur. Ainsi, lorsque nous agrandissons l'image, l'ordinateur est capable de dessiner avec précision les détails d'une image qu'il n'aurait pas été possible de produire avec une image pixelisée. Le pdf est un format de sauvegarde vectoriel.

Importer les fonctionnalités PDF

Dans les exemples précédents, nous avons employé `import processing.pdf.*` qui permet d'importer la librairie relative à la création de document PDF. Une librairie est un ensemble de fonctionnalités qui étend les possibilités de *Processing* et chaque fois que nous aurons à utiliser l'export PDF dans notre sketch, nous utiliserons cette commande d'importation. Si jamais vous essayez d'utiliser les méthodes relatives à l'export PDF sans avoir importé la librairie, vous allez rencontrer des erreurs qui s'afficheront dans la console et le programme ne pourra démarrer. Au lieu de taper cette ligne de code, vous pouvez l'ajouter automatiquement en cliquant dans le menu **Sketch > Import Library... > PDF Export**.

8.3 La vidéo

Processing vous offre la possibilité d'exporter votre sketch animé ou interactif sous forme d'animation vidéo au format QuickTime. Pour créer des vidéos, nous allons utiliser un objet prédéfini *MovieMaker* de la librairie *org.processing.video*. Attention, pour les utilisateurs sous Linux, cette fonction n'est à ce jour pas disponible.

L'objet *MovieMaker*

L'objet *MovieMaker* est une classe prédéfinie de *Processing* qui permet de créer un fichier vidéo, et de lui ajouter des images au fur et à mesure de l'exécution de votre programme. Pour pouvoir l'utiliser dans votre sketch, il faut importer la librairie vidéo de *Processing*. Il faut ensuite déclarer une variable en en-tête de notre programme qui stockera notre objet vidéo. Dans le *setup()* du sketch, il faut créer l'objet en lui donnant un certain nombre de paramètres :

- Une référence au sketch
- La largeur de la vidéo
- La hauteur de la vidéo
- Le nom du fichier de sortie

```
import processing.video.*;
//Déclaration de la variable qui recevra notre objet
MovieMaker mm;
```

```
void setup() {
    size(300,300);
```

```
    // Création de l'objet MovieMaker.
    // La vidéo aura la même taille que notre esquisse .
    mm = new MovieMaker(this, width, height, "Esquisse.mov");
}
```

Ajouter des images

A chaque appel de la méthode *draw()* de *Processing*, il faut indiquer à l'objet *MovieMaker* qu'il doit enregistrer l'image actuellement à l'écran dans le fichier vidéo.

```
void draw() {
    ellipse(mouseX,mouseY,20,20); //On dessine une ellipse
    mm.addFrame(); //On ajoute une image à la vidéo
    mm.addFrame(); //On ajoute une image à la vidéo
}
```

Finaliser la vidéo

A ce stade, l'enregistrement de la vidéo se termine lorsque l'on quitte le programme. On peut indiquer à l'objet *MovieMaker* d'arrêter l'enregistrement à tout moment. Dans notre exemple, nous allons ajouter une action qui stoppe l'enregistrement vidéo lorsqu'on appuie sur n'importe quelle touche du clavier de votre ordinateur.

```
void keyPressed() {
    mm.finish(); }
```

Personnaliser l'enregistrement

Par défaut, l'objet `MovieMaker` enregistre une vidéo non compressée à 15 images par seconde. Pour des applications interactives nécessitant un temps de réaction très rapide de l'ordinateur, cette qualité d'enregistrement peut s'avérer trop lourde. À l'inverse, pour d'autres types de création, on peut avoir besoin de produire des vidéos encore plus fluides (comportant davantage d'images par seconde). Avec l'objet `MovieMaker`, il est possible de personnaliser les caractéristiques de l'enregistrement en précisant notamment un format particulier ainsi que le nombre d'images par seconde. Les paramètres sont les suivants :

- Une référence au sketch
- La largeur de la vidéo
- La hauteur de la vidéo
- Le nom du fichier de sortie

- Nombre d'images par seconde
- Codec de compression
- Qualité

```
mm = new MovieMaker(this, width, height, "sketch.mov", 30, Movie-  
Maker.H263, MovieMaker.HIGH);
```

Avec ces paramètres, l'enregistrement de la vidéo est beaucoup plus fluide. Voici quelques autres codecs vidéo intéressants :

- `MovieMaker.H264` : Lent à encoder et décoder. Occupe par contre très peu d'espace-disque sur votre ordinateur.
- `MovieMaker.MOTION_JPEG_B` : Très rapide à encoder et décoder. Occupe également peu d'espace-disque.
- `MovieMaker.ANIMATION` : Supporte le canal alpha permettant la gestion de la transparence, très utile pour certaines images. Occupe énormément d'espace-disque, la vidéo étant compressée sans perte.

Erreurs & Astuces

Sommaire

A.1 Les erreurs courantes	121
A.2 Les astuces	124

A.1 Les erreurs courantes

Processing affiche un certain nombre de messages d'erreur dans la bande grise de la console (qui dans ce cas de figure change de couleur en devenant rouge foncé). Voici une liste des erreurs les plus courantes.

Unexpected token

Cette erreur se produit fréquemment lorsque vous oubliez un ";" en fin de ligne ou que vous n'avez pas correctement fermé un bloc d'accolades. Le programme voit une instruction qui fait deux lignes et dont la syntaxe est par conséquent erronée.

```
int nombreEntier = 1
float nombreVirgule = 0.1;
```

Il apparaît alors l'erreur : "unexpected token : int".

Le code correct est :

```
int nombreEntier = 1;
float nombreVirgule = 0.1;
```

Cannot find anything named

Cette erreur se produit lorsque vous appelez une variable qui n'existe pas. Vérifiez que vous avez bien déclaré votre variable et que vous avez respecté la portée des variables.

```
int nombre2 = nombre1 + 5;
```

Il apparaît alors l'erreur : "Cannot find anything named "nombre 1".

Le code correct est :

```
int nombre1 = 10;  
int nombre2 = nombre1 + 5;
```

Found one too many...

Cette erreur se produit lorsque vous n'avez pas correctement fermé un bloc par une accolade.

```
void draw() {  
}  
void setup() {
```

Il apparaît alors l'erreur : "Found one too many { characters without a } to match it". Le code correct est :

```
void draw() {  
}  
void setup() {  
}
```

ArithmeticException : / by zero

```
int diviseur = 0;  
println(1 / diviseur);
```

Il apparaît alors l'erreur : " ArithmeticException : / by zero ". Dans *Processing*,

on ne peut jamais diviser un nombre par zéro. Au besoin, vérifier que le diviseur n'est pas zéro avant de faire la division. C'est la fonction du morceau de code suivant :

```
int diviseur = 0;  
if (diviseur != 0) {  
    println(1 / diviseur);  
} else {  
    println("On ne peut pas diviser par zéro!");  
}
```

Cannot convert ... to ...

Cette erreur survient quand vous essayez d'affecter une valeur qui ne correspond pas au type d'une variable.

```
int a = 10;  
float b = 10.5;  
  
a = b;
```

Il apparaît alors l'erreur : " cannot convert from float to int ". Dans le cas présent, pour que nous puissions affecter un nombre à virgule à une variable qui est prévue pour stocker un nombre entier, il faut forcer *Processing* à la transformer. Le code correct est :

```
int a = 10;  
float b = 10.5;
```

```
a = int(b);
```

ArrayIndexOutOfBoundsException

Cette erreur survient lorsque vous tentez d'accéder à un élément en dehors des limites d'un tableau. Dans l'exemple suivant, le tableau a une taille de 3 (case 0, 1 et 2) et nous essayons d'accéder à la case 4.

```
int[] nombres = new int[3];
nombres[0] = 1;
nombres[1] = 20;
nombres[2] = 5;
println(nombres[4]);
```

Il apparaît alors l'erreur : "ArrayIndexOutOfBoundsException : 4". Le code correct est le suivant (la 3e case correspond au rang 2 et non pas au rang 3 ou 4 puisqu'un tableau commence toujours à partir d'une case de rang 0) :

```
int[] numbers = new int[3];
numbers[0] = 1;
numbers[1] = 20;
numbers[2] = 5;
println(numbers[2]);
```

NullPointerException

Cette erreur se produit lorsque vous tentez d'accéder à un élément qui n'existe pas en mémoire. Par exemple en essayant d'accéder à un objet qui n'a pas encore été initialisé.

```
Balle maBalle;
void setup() {
    maBalle.x = 10;
}
```

```
class Balle {
    int x;
    int y;
}
```

Il apparaît alors l'erreur : "NullPointerException". Le code correct est :

```
Balle maBalle = new Balle();
void setup() {
    maBalle.x = 10;
}
```

```
class Balle {
    int x;
    int y;
}
```

A.2 Les astuces

Pour vous éviter certaines erreurs de base et aller plus loin dans l'utilisation de *Processing*, ce chapitre présente plusieurs astuces et morceaux de code qui permettent de résoudre une série de problèmes fréquemment rencontrés. N'hésitez pas à les utiliser dans vos programmes.

Augmenter la mémoire

Si vous créez des sketches qui utilisent de la vidéo, de nombreuses images ou qui travaillent avec des données volumineuses, il peut arriver que votre programme génère une erreur pour cause de mémoire insuffisante. Voici l'erreur affichée par *Processing* dans la console : "OutOfMemory : You may need to increase the memory setting in Preferences". Pour résoudre ce problème, vous devez ouvrir les préférences de *Processing* et changer la quantité de mémoire allouée à votre sketch.

Inverser la valeur d'un booléen (vrai/faux)

Voici un mécanisme permettant d'inverser la valeur d'un booléen sans avoir à tester à l'aide de la condition `if/ then` si sa valeur est vraie ou fausse. Cette astuce repose sur le fait qu'en précédant la valeur booléenne d'un point d'exclamation, on obtient son contraire.

```
boolean afficheInfo;
void setup() {
  afficheInfo = false;
}
void draw() {
```

```
  background(0); // Si vrai, on affiche le texte
  if (afficheInfo) text("Info", 10, 20);
}
```

```
void keyPressed() {
  // on affecte à afficheInfo l'inverse de sa valeur actuelle
  afficheInfo = ! afficheInfo;
}
```

Intervalomètre avec `frameRate()`, `frameCount` et modulo `%`

Si l'on souhaite effectuer une action à intervalle régulier, le procédé le plus simple consiste à utiliser les méthodes de comptage d'images de *Processing* : `frameRate()`, `frameCount` ainsi que la commande `%` (modulo). Notez que cette astuce ne permet de gérer qu'un seul intervalle de temps par programme. Détaillons chacune de ces fonctions :

- la variable `frameCount` permet de connaître le nombre de fois où `draw()` a été appelé depuis le lancement du programme.
- l'instruction `frameRate()` fixe la fréquence des appels à `draw()` par seconde (par défaut 30).
- l'opérateur `%` (modulo) retourne le reste de la division de deux nombres, ce reste étant strictement inférieur au nombre diviseur (c'est une règle mathématique!). Ainsi `7%3` retourne 1 ou encore `8%2` retourne 0. L'opérateur modulo permet de compter sans jamais dépasser un certain nombre (la base du modulo).

Ainsi, si l'on veut effectuer une action toutes les secondes, on la déclenchera dans le cas où le reste de la division de `frameCount` par la valeur passée à `frameRate()` est bien égal à zéro. Dans l'exemple qui suit, nous changeons la couleur de fond à chaque seconde en vérifiant que notre compteur est un

multiple de 24.

```
color couleur;  
void setup() {  
  couleur = 0;  
  frameRate(24);  
}  
  
void draw() {  
  background(couleur);  
  if (frameCount % 24 == 0) {  
    couleur = (int) random(255);  
  }  
}
```

Exercice : modifiez le code ci-dessus pour que le changement de couleur ait lieu toutes les 3 secondes.

Créer une classe intervalomètre

Dans l'astuce précédente, nous avons vu comment créer un intervalomètre simple, très utile pour exécuter une action à intervalle régulier, mais qui comporte certaines limitations. A présent, nous allons apprendre à programmer un intervalomètre gérant plusieurs intervalles de temps au sein d'un même sketch, et ce, avec précision (sans variation de vitesse lorsque votre ordinateur est occupé à exécuter simultanément de nombreuses tâches). A titre d'illustration, nous allons créer un objet `Intervalometre` chargé d'exécuter un bloc de code à un intervalle régulier. Le système de comptage fonctionne sur le principe suivant : mémoriser le moment auquel l'objet a été exécuté pour la dernière fois et à chaque appel de la méthode `draw()` de vérifier si le temps actuel est

plus grand que ce moment de temps antérieur auquel on a ajouté un intervalle donné. Nous allons nous aider de la méthode `millis()` qui retourne le nombre de millisecondes depuis le lancement du programme. Notre classe sera composée de deux caractéristiques et d'une action :

- L'intervalle de temps (un intervalle que vous aurez déterminé à votre convenance)
- Le temps de la dernière vérification de l'intervalomètre
- Une action qui vérifie si le prochain intervalle a été franchi.

```
class Intervalometre {  
  int intervalle;  
  int dernier_tic;  
  Intervalometre(int intervalle_initial) {  
    intervalle = intervalle_initial;  
    dernier_tic = millis();  
  }  
  boolean verifierIntervalle() {  
    if (millis() > dernier_tic + intervalle) {  
      dernier_tic = millis();  
      return true;  
    } else {  
      return false;  
    }  
  }  
}
```

Ici, la méthode `verifierIntervalle()` vérifie si l'intervalle de temps est écoulé depuis la dernière fois. Elle retourne vrai ou faux. En d'autres termes,

cela revient à décider d'exécuter une certaine action à chaque fois que notre horloge fait tic. La dernière étape de programmation de notre sketch va consister à déclarer une variable qui stockera notre intervalomètre. Chaque appel de la méthode `draw()` exécutera `verifierIntervalle()` de notre objet `Intervalometre`. Si cette méthode nous retourne vrai, nous allons faire apparaître un rectangle positionné au hasard sur l'écran. L'intervalle de temps choisi dans cet exemple est de 100 millisecondes.

```
Intervalometre intervalometre;
void setup() {
  intervalometre = new Intervalometre(100);
}

void draw() {
  if (intervalometre.verifierIntervalle()) {
    rect(random(0, width), random(0, height), 10, 10);
  }
}
```

Suspendre la répétition de `draw()`

Tant que l'on n'a pas quitté le programme, et sauf indication expresse de notre part, la fonction `draw()` est appelée en boucle durant toute l'exécution du programme. Il existe cependant un moyen d'interrompre ce cycle à l'aide de la fonction `noLoop()`. La fonction `loop()` a l'effet inverse. L'exemple suivant montre comment interrompre et reprendre ce mouvement à l'aide de la touche 's' et de la barre d'espace.

```
int position = 0;
int increment = 1;
void setup() {
}

void draw() {
  background(0);
  ellipse(50, position, 5, 5);
  // on déplace le point de 1 pixel
  position += increment;
  // si on touche les bords
  if (position > height || position < 0) {
    // alors on inverse la valeur d'incrément
    increment *= -1;
  }
}

void keyPressed() {
  if (key == ' ') {
    // on relance le cycle
    loop();
  } else if (key == 's' || key == 'S') {
    // on arrête le cycle
    noLoop();
  }
}
```

Quitter le programme

Pour quitter un programme en utilisant du code plutôt qu'une intervention utilisateur, vous pouvez appeler la méthode `exit()`.

```
// Vous pouvez ajouter ce code à l'exemple précédent
else if (key == 'e' || key == 'E') {
    exit(); // on quitte le programme
}
```


Pour étendre les fonctionnalités de *Processing*

B.1 Les librairies externes

Une librairie, c'est une collection de classes que l'on peut réutiliser dans chacun de nos projets. Mise à disposition sur Internet, une multitude de librairies intéressantes existe pour *Processing*. Pour pouvoir en utiliser une, il suffit de la télécharger de son site internet, de décompresser l'archive et de le placer dans le dossier `libraries`. Le dossier `libraries` se trouve dans le dossier de travail de *Processing* (menu Sketch puis Show Sketch Folder). Par ailleurs, plusieurs librairies sont déjà installées par défaut dans ce dossier lors de l'installation de *Processing*.

Dans tous les cas, pour les utiliser dans un de vos sketches, choisir son nom dans le menu Sketch > Library. Ces librairies sont libres : les utilisateurs ont la liberté d'exécuter, de copier, de distribuer, d'étudier, de modifier et d'améliorer leur code.

Où trouver des librairies

La majorité des librairies ne sont pas installées par défaut avec *Processing*, mais doivent plutôt être téléchargées via le site web de leur auteur. Cette pratique permet d'obtenir en tout temps la version la plus à jour d'une

librairie et simplifie grandement sa distribution. On peut, sur le site de *Processing*, accéder à un annuaire des principales librairies à l'adresse suivante : <http://processing.org/reference/libraries/>.

Installer une librairie

Nous allons maintenant installer une librairie que nous téléchargeons de l'internet. Nous utiliserons la librairie *ShapeTween*, car celle-ci est très simple à utiliser. Elle fournit une variété de moyens d'animer des éléments. Ce genre d'outil est fort utile pour créer des animations fluides.

Pour désigner des animations et des transitions, le mot *tween* est souvent utilisé. Ce mot vient de l'anglais *in between* et désigne les images que l'on dessine entre chacune des images-clés dans le cinéma d'animation traditionnel. Il s'agit donc d'interpolation de mouvements. L'interpolation consiste à déduire des positions intermédiaires entre des points connus.

On peut trouver un lien vers la page du projet *ShapeTween* à partir de la page *Libraries* sur le site de *Processing* à l'adresse <http://www.leebyron.com/else/shapetween/>. Sur cette page, vous trouverez le lien intitulé *shapetween.zip*. Téléchargez l'archive et décompressez le fichier zip. Placez son contenu dans le répertoire `libraries` de votre répertoire de sketch. Vous pouvez

ouvrir ce dossier en utilisant le menu *Sketch > Show sketch folder*. Le dossier *libraries* se trouve dans le même dossier que vos *sketchs*. Vous devez y placer le répertoire extrait de l'archive téléchargée ayant le nom de la librairie. Si ce dossier *libraries* n'existe pas déjà, créez le tout simplement. Une fois ces opérations effectuées, quittez l'application *Processing* et lancez-la à nouveau.

Essayer un exemple

Une bonne librairie fournit également une documentation de chacune des fonctionnalités et instructions offertes (classes, méthodes et attributs). Elle comporte également le plus souvent des exemples. Si c'est le cas, il suffit d'y accéder en parcourant le menu *File > Sketchbook > Examples*. Le nom de notre librairie devrait s'y trouver. Notez que cette procédure fonctionne uniquement sur Mac OS X et Windows. Sur GNU/Linux, les exemples des librairies installées par nous-mêmes ne s'affichent pas dans ce menu.

Si la librairie ne contient pas d'exemples, nous pouvons essayer d'en trouver sur la Toile afin de le copier-coller directement de la page du site vers la fenêtre d'édition de *Processing*. Comme exercice, nous allons copier-coller un exemple présenté sur le site internet de *ShapeTween*. Sur le site Web de la librairie, on peut voir une liste d'exemples sous la rubrique *Example Code*. Cliquez sur *Basic Tween* afin de voir un exemple très simple de mouvement contrôlé à l'aide de cette librairie. Copiez le code proposé sur la page et collez-le dans votre fenêtre d'édition de *Processing*. Avec la librairie *ShapeTween*, il est également possible de créer des mouvements qui accélèrent, puis décélèrent, mais les autres exemples proposés sur le site de cette librairie contiennent des erreurs au moment d'écrire ces lignes (le répertoire consacré aux exemples comporte la mention "en cours de construction"). Après avoir vérifié que l'exemple proposé fonctionne, vous pouvez modifier l'exemple, le simplifier et l'utiliser comme base pour des développements ultérieurs.

```
import megamu.shapetween.*;
Tween ani ;

void setup() {
  size( 200, 200 );
  ani = new Tween(this, 2, Tween.SECONDS);
  ani.start();
}

void draw() {
  background(200);

  float x1 = 20;
  float y1 = 30;
  float x2 = 180;
  float y2 = 190;

  float x = lerp( x1, x2, ani.position() );
  float y = lerp( y1, y2, ani.position() );

  fill(0);
  ellipse( x, y, 8, 8 );
}

void mousePressed() {
  ani.start();
}
```

Les librairies propriétaires

Parfois on télécharge une librairie et on constate que l'archive ne contient pas de code source (des fichiers Java). Puis parcourant le site web, on voit soudain que celui-ci contient la mention "Tous droits réservés", sans plus d'explication. Dans ce cas, cela signifie que la librairie n'est pas libre : elle est propriétaire. Vous pouvez l'utiliser gratuitement, mais sans avoir accès à son code ! Vous ne pouvez pas la modifier, ni étudier son fonctionnement. Il

se peut fort bien qu'elle ne soit pas compatible avec la prochaine version de *Processing*, et dans ce cas, vous ne serez pas en mesure de régler ce problème.

Si ces considérations sont importantes pour vous, soyez vigilants. Vérifiez toujours que le code source est inclus avec une librairie, et que son code comporte bien un en-tête avec la mention d'une licence libre. Le site de la Free Software Foundation fournit une liste de licences libres compatibles avec la GPL <http://www.gnu.org/licenses/license-list.fr.html>.

Index

éclairage, 17
épaisseur, 32
3D, 17, 45

animation, 75
arc, 15
audio, 93, 106

boîte, 18
bordure, 32
boucle for, 53

caméra, 110
cercle, 14
clavier, 99
collision, 84
commentaire, 11, 73
compteur, 54, 81
condition, 51
console, 11
contour, 17
couleur, 19
couleur de contour, 20
couleur de fond, 19

couleur de remplissage, 19
courbe, 15
curseur, 104

déplacer, 38, 84
degrees, 39
dossier de travail, 10
draw(), 75

ellipse, 14
erreur, 121
exportation, 113

fondu, 78
forme, 13
forme libre, 16

horloge, 95

image, 25, 59
impression, 115
interactif, 75, 99
interface, 9

langage, 9

librairie, 129
librairie externe, 129
ligne, 14
lissage, 32
liste, 57

mémoire, 124
méthode, 63
méthodes, 11
mettre à l'échelle, 40
microphone, 106
Minim, 93

objet, 69
opérations arithmétiques, 11

PDF, 115
pixel, 31, 60
point, 13
point virgule, 10

quadrilatère, 15

répétition, 53
radians, 39
rectangle, 14
redimensionner, 40
remplissage, 17

setup(), 64

sketch, 10
son, 93, 107
souris, 102
sphère, 18

temps, 80
texte, 23
tourner, 39
transformation, 38
triangle, 15
typographie, 34

variable, 47
variable boolean, 48
variable char, 49
variable color, 49
variable double, 48
variable float, 48
variable int, 48
variable string, 49
vidéo, 110, 118

width, height, 12

